ArkTS Specification

Release 7.0.1

2026.04.14

CONTENTS

1 Introduction **1**
- 1.1 Overall Description . . . 1
- 1.2 Lexical and Syntactic Notation . . . 2
- 1.3 Terms and Definitions . . . 3

2 Lexical Elements **9**
- 2.1 Use of Unicode Characters . . . 9
- 2.2 Lexical Input Elements . . . 9
- 2.3 White Spaces . . . 9
- 2.4 Line Separators . . . 10
- 2.5 Tokens . . . 10
- 2.6 Identifiers . . . 11
- 2.7 Keywords . . . 12
- 2.8 Operators and Punctuators . . . 13
- 2.9 Literals . . . 14
  - 2.9.1 Numeric Literals . . . 14
  - 2.9.2 Integer Literals . . . 14
  - 2.9.3 Floating-Point Literals . . . 17
  - 2.9.4 Bigint Literals . . . 18
  - 2.9.5 Boolean Literals . . . 18
  - 2.9.6 String Literals . . . 19
  - 2.9.7 Multiline String Literal . . . 20
  - 2.9.8 `Undefined` Literal . . . 21
  - 2.9.9 `Null` Literal . . . 21
- 2.10 Comments . . . 22
- 2.11 Semicolons . . . 22

3 Types **23**
- 3.1 Predefined Types . . . 24
- 3.2 User-Defined Types . . . 24
- 3.3 Using Types . . . 25
- 3.4 Named Types . . . 27
- 3.5 Type References . . . 27
- 3.6 Value Types . . . 28
  - 3.6.1 Numeric Types . . . 28
  - 3.6.2 Integer Types and Operations . . . 29
  - 3.6.3 Floating-Point Types and Operations . . . 30
  - 3.6.4 Type `boolean` . . . 32
- 3.7 Reference Types . . . 32
- 3.8 Type `Any` . . . 33

3.9 Type `Object` . . . 33
3.10 Type `never` . . . 33
3.11 Type `void` or `undefined` . . . 34
3.12 Type `null` . . . 35
3.13 Type `string` . . . 36
3.14 Type `bigint` . . . 36
3.15 Literal Types . . . 37
3.15.1 String Literal Types . . . 38
3.16 Array Types . . . 38
3.16.1 Resizable Array Types . . . 38
3.16.2 Readonly Array Types . . . 40
3.17 Tuple Types . . . 40
3.17.1 Readonly Tuple Types . . . 41
3.17.2 Type `Tuple` . . . 41
3.18 Function Types . . . 42
3.18.1 Type `Function` . . . 44
3.19 Union Types . . . 45
3.19.1 Union Types Normalization . . . 46
3.19.2 Access to Common Union Members . . . 47
3.19.3 `Keyof` Types . . . 48
3.20 Nullish Types . . . 49
3.21 Utility Types . . . 50
3.21.1 Awaited Utility Type . . . 50
3.21.2 NonNullable Utility Type . . . 51
3.21.3 Partial Utility Type . . . 52
3.21.4 Required Utility Type . . . 53
3.21.5 Readonly Utility Type . . . 54
3.21.6 Record Utility Type . . . 54
3.21.7 ReturnType Utility Type . . . 55
3.21.8 Utility Type Private Fields . . . 55
3.21.9 Nesting Utility Types . . . 56
3.22 Default Values for Types . . . 56

4 Names, Declarations and Scopes **59**
4.1 Names . . . 59
4.2 Declarations . . . 60
4.2.1 Declaration Distinguishable by Signatures . . . 61
4.3 Scopes . . . 62
4.4 Accessible . . . 63
4.5 Type Declarations . . . 63
4.5.1 Type Alias Declaration . . . 64
4.6 Variable and Constant Declarations . . . 66
4.6.1 Variable Declarations . . . 66
4.6.2 Constant Declarations . . . 67
4.6.3 Validity of Initializer . . . 67
4.6.4 Assignability with Initializer . . . 68
4.6.5 Type Inference from Initializer . . . 68
4.7 Function Declarations . . . 69
4.7.1 Signatures . . . 70
4.7.2 Parameter List . . . 70
4.7.3 Readonly Parameters . . . 71
4.7.4 Optional Parameters . . . 71
4.7.5 Rest Parameter . . . 72
4.7.6 Shadowing by Parameter . . . 75

4.7.7 Return Type . . . 75

5 Generics **77**
5.1 Type Parameters . . . 77
5.1.1 Type Parameter Constraint . . . 78
5.1.2 Type Parameter Default . . . 79
5.1.3 Type Parameter Variance . . . 80
5.2 Generic Instantiations . . . 81
5.2.1 Type Arguments . . . 81
5.2.2 Explicit Generic Instantiations . . . 82
5.2.3 Implicit Generic Instantiations . . . 83

6 Contexts and Conversions **85**
6.1 Assignment-like Contexts . . . 85
6.2 String Operator Contexts . . . 86
6.3 Numeric Operator Contexts . . . 87
6.3.1 Numeric Conversions for Relational and Equality Operands . . . 87
6.3.2 `char` Conversions for Relational and Equality Operands . . . 88
6.4 Implicit Conversions . . . 88
6.4.1 Widening Numeric Conversions . . . 88
6.4.2 Widening Numeric to a Union Type . . . 89
6.4.3 Enumeration to Numeric Type Conversion . . . 90
6.4.4 Enumeration to `string` Type Conversion . . . 90
6.5 Numeric Casting Conversions . . . 90

7 Expressions **93**
7.1 Operators . . . 94
7.1.1 Operator Precedence . . . 95
7.2 Evaluation of Expressions . . . 96
7.2.1 Type of Expression . . . 96
7.2.2 Normal and Abrupt Completion of Expression Evaluation . . . 97
7.2.3 Order of Expression Evaluation . . . 98
7.2.4 Evaluation of Arguments . . . 99
7.2.5 Evaluation of Other Expressions . . . 100
7.3 Literal . . . 100
7.4 Named Reference . . . 100
7.4.1 Function Reference . . . 101
7.4.2 Method Reference . . . 102
7.5 Array Literal . . . 103
7.5.1 Array Literal Type Inference from Context . . . 104
7.5.2 Array Type Inference from Types of Elements . . . 106
7.6 Object Literal . . . 107
7.6.1 Object Literal of Class Type . . . 108
7.6.2 Object Literal of Interface Type . . . 111
7.6.3 Object Literal of `Record` Type . . . 113
7.6.4 Object Literal Evaluation . . . 114
7.7 Spread Expression . . . 114
7.8 Parenthesized Expression . . . 118
7.9 `this` Expression . . . 118
7.10 Field Access Expression . . . 120
7.10.1 Accessing Static Fields . . . 120
7.10.2 Accessing Current Object Fields . . . 120
7.10.3 Accessing SuperClass Accessors . . . 121
7.11 Method Call Expression . . . 121

7.11.1 Step 1: Selection of Type to Use . . . 122
7.11.2 Step 2: Selection of Entity to Call . . . 122
7.11.3 Step 3: Checking Modifiers . . . 122
7.11.4 Type of Method Call Expression . . . 123
7.12 Function Call Expression . . . 123
7.12.1 Call Arguments . . . 124
7.13 Indexing Expressions . . . 125
7.13.1 Array Indexing Expression . . . 125
7.13.2 String Indexing Expression . . . 126
7.13.3 Record Indexing Expression . . . 127
7.14 Chaining Operator . . . 128
7.15 `New` Expressions . . . 130
7.16 `instanceof` Expression . . . 131
7.17 `Cast` Expression . . . 132
7.17.1 Type Inference in Cast Expression . . . 133
7.17.2 Runtime Checking in Cast Expression . . . 134
7.18 `typeof` Expression . . . 135
7.19 Ensure-Not-Nullish Expression . . . 137
7.20 Nullish-Coalescing Expression . . . 138
7.21 Unary Expressions . . . 139
7.21.1 Postfix Increment . . . 139
7.21.2 Postfix Decrement . . . 140
7.21.3 Prefix Increment . . . 141
7.21.4 Prefix Decrement . . . 141
7.21.5 Unary Plus . . . 142
7.21.6 Unary Minus . . . 142
7.21.7 Bitwise Complement . . . 143
7.21.8 Logical Complement . . . 144
7.22 Binary Expressions . . . 144
7.23 Multiplicative Expressions . . . 146
7.23.1 Multiplication . . . 147
7.23.2 Division . . . 148
7.23.3 Remainder . . . 149
7.24 Exponentiation Expression . . . 150
7.25 Additive Expressions . . . 152
7.25.1 String Concatenation . . . 152
7.25.2 Additive Operators for Numeric Types . . . 153
7.26 Shift Expressions . . . 154
7.27 Relational Expressions . . . 155
7.27.1 Numeric Relational Operators . . . 155
7.27.2 Bigint Relational Operators . . . 156
7.27.3 String Relational Operators . . . 157
7.27.4 Boolean Relational Operators . . . 157
7.27.5 `char` Relational Operators . . . 157
7.27.6 Enumeration Relational Operators . . . 157
7.28 Equality Expressions . . . 158
7.28.1 Numeric Equality Operators . . . 160
7.28.2 Function Type Equality Operators . . . 161
7.28.3 Extended Equality with `null` or `undefined` . . . 161
7.29 Bitwise and Logical Expressions . . . 162
7.29.1 Integer Bitwise Operators . . . 162
7.29.2 Boolean Logical Operators . . . 162
7.30 Conditional-And Expression . . . 163
7.31 Conditional-Or Expression . . . 163

7.32 Assignment . . . 164
7.32.1 Simple Assignment Operator . . . 165
7.32.2 Compound Assignment Operators . . . 166
7.32.3 Left-Hand-Side Expressions . . . 168
7.33 Ternary Conditional Expressions . . . 168
7.34 String Interpolation Expressions . . . 169
7.35 Lambda Expressions . . . 170
7.35.1 Lambda Signature . . . 171
7.35.2 Lambda Body . . . 172
7.35.3 Lambda Expression Type . . . 172
7.35.4 Runtime Evaluation of Lambda Expressions . . . 172
7.36 Constant Expressions . . . 174
7.36.1 Specifics of Constant Expressions Evaluation . . . 175

8 Statements **177**
8.1 Normal and Abrupt Statement Execution . . . 178
8.2 Expression Statements . . . 178
8.3 Block . . . 179
8.4 Constant Or Variable Declarations . . . 179
8.5 `if` Statements . . . 180
8.6 Loop Statements . . . 181
8.7 `while` Statements and `do` Statements . . . 182
8.8 `for` Statements . . . 182
8.9 `for-of` Statements . . . 183
8.10 `break` Statements . . . 184
8.11 `continue` Statements . . . 185
8.12 `return` Statements . . . 186
8.13 `switch` Statements . . . 187
8.14 `throw` Statements . . . 188
8.15 `try` Statements . . . 189
8.15.1 `catch` Clause . . . 189
8.15.2 `finally` Clause . . . 190
8.15.3 `try` Statement Execution . . . 191

9 Classes **193**
9.1 Class Declarations . . . 193
9.1.1 Abstract Classes . . . 194
9.2 Class Extension Clause . . . 195
9.3 Class Implementation Clause . . . 196
9.3.1 Implementing Required Interface Properties . . . 198
9.3.2 Implementing Optional Interface Properties . . . 203
9.4 Class Members . . . 204
9.5 Access Modifiers . . . 206
9.5.1 Private Access Modifier . . . 206
9.5.2 Protected Access Modifier . . . 207
9.5.3 Public Access Modifier . . . 208
9.6 Field Declarations . . . 208
9.6.1 Static and Instance Fields . . . 209
9.6.2 Readonly (Constant) Fields . . . 209
9.6.3 Optional Fields . . . 210
9.6.4 Field Initialization . . . 210
9.6.5 Fields with Late Initialization . . . 211
9.6.6 Override Fields . . . 211
9.7 Method Declarations . . . 214

9.7.1 Static Methods . . . 215
9.7.2 Instance Methods . . . 216
9.7.3 Abstract Methods . . . 216
9.7.4 Async Methods . . . 217
9.7.5 Overriding Methods . . . 217
9.7.6 Native Methods . . . 217
9.7.7 Method Body . . . 217
9.7.8 Methods Returning `this` . . . 218
9.8 Class Accessor Declarations . . . 219
9.9 Constructor Declaration . . . 221
9.9.1 Constructor Body . . . 222
9.9.2 Explicit Constructor Call . . . 226
9.9.3 Default Constructor . . . 227
9.10 Inheritance . . . 227

10 Interfaces **229**
10.1 Interface Declarations . . . 229
10.2 Superinterfaces and Subinterfaces . . . 230
10.3 Interface Members . . . 231
10.4 Interface Properties . . . 231
10.4.1 Required Interface Properties . . . 232
10.4.2 Optional Interface Properties . . . 232
10.5 Interface Method Declarations . . . 233
10.6 Interface Inheritance . . . 233

11 Enumerations **235**
11.1 Enumeration Base Type . . . 236
11.2 Enumeration with Explicit Base Type . . . 237
11.3 Initialization of Enumeration Members . . . 237
11.4 Enumeration Methods . . . 238

12 Error Handling **241**
12.1 Errors . . . 241

13 Namespaces and Modules **243**
13.1 Module Declarations . . . 243
13.2 Module Header . . . 244
13.3 Namespace Declarations . . . 245
13.4 Import Directives . . . 249
13.4.1 Bind All with Qualified Access . . . 250
13.4.2 Default Import Binding . . . 251
13.4.3 Selective Binding . . . 251
13.4.4 Import Type Directive . . . 253
13.4.5 Import Path . . . 254
13.4.6 Several Bindings for One Import Path . . . 255
13.5 Top-Level Declarations . . . 256
13.6 Exported Declarations . . . 257
13.7 Export Directives . . . 262
13.7.1 Selective Export Directive . . . 263
13.7.2 Single Export Directive . . . 263
13.7.3 Export Type Directive . . . 264
13.7.4 Re-Export Directive . . . 264
13.8 Top-Level Statements . . . 265
13.9 Multifile Module . . . 266
13.10 Standard Library Usage . . . 268

13.11 Program Entry Point . . . 268

14 Ambient Declarations **271**
14.1 Ambient Constant or Variable Declarations . . . 272
14.2 Ambient Function Declarations . . . 272
14.3 Ambient Overload Function Declarations . . . 273
14.4 Ambient Class Declarations . . . 273
14.4.1 Ambient Indexer . . . 275
14.4.2 Ambient Call Signature . . . 275
14.4.3 Ambient Iterable . . . 276
14.5 Ambient Interface Declarations . . . 277
14.6 Ambient Enumeration Declarations . . . 278
14.7 Ambient Namespace Declarations . . . 278
14.7.1 Implementing Ambient Namespace Declaration . . . 280
14.8 Ambient Accessor Declarations . . . 280

15 Semantic Rules **281**
15.1 Semantic Essentials . . . 281
15.1.1 Type of Standalone Expression . . . 281
15.1.2 Specifics of Assignment-like Contexts . . . 282
15.1.3 Specifics of Variable Initialization Context . . . 282
15.1.4 Specifics of Numeric Operator Contexts . . . 283
15.1.5 Specifics of String Operator Contexts . . . 283
15.1.6 Other Contexts . . . 283
15.1.7 Specifics of Type Parameters . . . 283
15.1.8 Semantic Essentials Summary . . . 284
15.2 Subtyping . . . 284
15.2.1 Subtyping for Non-Generic Classes and Interfaces . . . 285
15.2.2 Subtyping for Generic Classes and Interfaces . . . 286
15.2.3 Subtyping for Literal Types . . . 289
15.2.4 Subtyping for Tuple Types . . . 290
15.2.5 Subtyping for Union Types . . . 290
15.2.6 Subtyping for Function Types . . . 291
15.2.7 Subtyping for Fixed-Size Array Types . . . 292
15.2.8 Subtyping for Difference Types . . . 293
15.3 Type Identity . . . 293
15.4 Assignability . . . 294
15.5 Invariance, Covariance and Contravariance . . . 294
15.6 Compatibility of Call Arguments . . . 295
15.7 Type Inference . . . 297
15.7.1 Type Inference for Constant Expressions . . . 297
15.7.2 Return Type Inference . . . 299
15.8 Overriding . . . 300
15.8.1 Overriding in Classes . . . 300
15.8.2 Overriding in Interfaces . . . 302
15.8.3 Override-Compatible Signatures . . . 303
15.9 Overloading . . . 306
15.9.1 Implicit Function Overloading . . . 307
15.9.2 Implicit Method Overloading . . . 308
15.9.3 Implicit Constructor Overloading . . . 309
15.9.4 Overload-Equivalent Signatures . . . 309
15.10 Overload Resolution . . . 310
15.10.1 Forming an Overload Set . . . 310
15.10.2 Overload Set for Functions . . . 311

15.10.3 Overload Set for Interface Methods . . . 311
15.10.4 Overload Set for Class Static Methods . . . 312
15.10.5 Overload Set for Class Instance Methods . . . 313
15.10.6 Overload Set for Constructors . . . 315
15.10.7 Overload Set Warning . . . 315
15.10.8 Overload Set at Method Call . . . 315
15.10.9 Overloading and Overriding . . . 316
15.10.10Dynamic resolution of method calls . . . 318
15.11 Type Erasure . . . 319
15.12 Static Initialization . . . 321
15.12.1 Static Initialization Safety . . . 322
15.13 Compatibility Features . . . 322
15.13.1 Extended Conditional Expressions . . . 322

16 Concurrency **325**
16.1 Execution model . . . 325
16.2 Overview of concurrency features . . . 326
16.3 Asynchronous execution . . . 326
16.3.1 Asynchronous Functions . . . 326
16.3.2 Asynchronous Lambdas . . . 327
16.3.3 Asynchronous Methods . . . 327
16.3.4 `await` expression . . . 328
16.3.5 `Promise` class . . . 329
16.4 Parallel Execution . . . 329
16.4.1 Parallel execution API . . . 329
16.4.2 EAWorker API . . . 330
16.4.3 Taskpool API . . . 330
16.5 Synchronization . . . 330
16.5.1 Asynchronous lock . . . 331
16.5.2 Asynchronous mutex . . . 332
16.5.3 Asynchronous read/write lock . . . 334
16.5.4 Asynchronous condition variable . . . 334
16.5.5 Atomic operations . . . 336
16.5.6 Additional entities and other notes . . . 336
16.6 API details and restrictions . . . 336
16.6.1 Using the asynchronous API . . . 336
16.6.2 Promise class API . . . 336
16.6.3 Unhandled Rejected Promises . . . 337
16.6.4 Error handling policy . . . 337
16.7 Runtime implementation details . . . 338
16.7.1 Scheduling rules . . . 338

17 Experimental Features **339**
17.1 Type `char` . . . 339
17.1.1 `char` Literals . . . 340
17.1.2 `char` Operations . . . 340
17.2 Fixed-Size Array Types . . . 341
17.2.1 Fixed-Size Array Creation . . . 342
17.3 Value Array Types . . . 342
17.4 Resizable Array Creation Expressions . . . 343
17.4.1 Runtime Evaluation of Array Creation Expressions . . . 345
17.5 Indexable Types . . . 345
17.6 Iterable Types . . . 347
17.7 Callable Types . . . 348

17.7.1 Callable Types with `$_invoke` Method . . . 349
17.7.2 Callable Types with `$_instantiate` Method . . . 349
17.8 Statements . . . 351
17.8.1 For-of Explicit Type Annotation . . . 352
17.9 Explicit Overload Declarations . . . 352
17.9.1 Explicit Function Overload . . . 354
17.9.2 Explicit Class Method Overload . . . 356
17.9.3 Explicit Interface Method Overload . . . 359
17.9.4 Explicit Overload Name Same As Method Name . . . 360
17.10 Native Functions and Methods . . . 362
17.10.1 Native Functions . . . 362
17.10.2 Native Methods . . . 362
17.10.3 Native Constructors . . . 363
17.11 Classes Experimental . . . 363
17.11.1 Final Classes . . . 363
17.11.2 Final Methods . . . 363
17.12 Default Interface Method Declarations . . . 363
17.13 Adding Functionality to Existing Types . . . 364
17.13.1 Functions with Receiver . . . 364
17.13.2 Receiver Type . . . 368
17.13.3 Function Types with Receiver . . . 368
17.13.4 Lambda Expressions with Receiver . . . 370
17.14 Trailing Lambdas . . . 371
17.15 Accessor Declarations . . . 373
17.16 Pattern Matching . . . 374
17.16.1 Destructuring Assignment . . . 375

18 Annotations **377**
18.1 Declaring Annotations . . . 377
18.1.1 Types of Annotation Fields . . . 378
18.2 Using Annotations . . . 379
18.2.1 Using Single Field Annotations . . . 381
18.3 Exporting and Importing Annotations . . . 381
18.4 Ambient Annotations . . . 382
18.5 Standard Annotations . . . 383
18.5.1 Retention Annotation . . . 383
18.5.2 Target Annotation . . . 384
18.6 Runtime Access to Annotations . . . 385

19 Standard Library **387**

20 Implementation Details **389**
20.1 Import Path Lookup . . . 389
20.2 Modules in Host System . . . 389
20.3 Getting Runtime Type Via Reflection . . . 389
20.4 Ensuring Module Initialization . . . 390
20.5 Generic and Function Types Peculiarities . . . 390
20.6 Keyword `struct` and ArkUI . . . 390
20.7 Make a Bridge Method for Overriding Method . . . 391

21 Runtime System **393**
21.1 Overview . . . 393
21.2 Fully Qualified Name . . . 393
21.3 Binary File Format . . . 393
21.3.1 Alignment . . . 394

21.3.2 Endianness . . . 394
21.3.3 Offsets . . . 394
21.3.4 Data Types . . . 394
21.3.5 File Layout . . . 395
21.4 Runtime Type System . . . 420
21.4.1 Subtyping . . . 421
21.5 Runtime Name . . . 421
21.6 Language Representation in Binary File . . . 422
21.6.1 Value Types . . . 422
21.6.2 Type `Any` . . . 424
21.6.3 Type `Object` . . . 424
21.6.4 Type `never` . . . 424
21.6.5 Type `void` . . . 424
21.6.6 Type `null` . . . 424
21.6.7 Type `string` . . . 425
21.6.8 Type `bigint` . . . 425
21.6.9 Array types . . . 425
21.6.10 Tuple Types . . . 425
21.6.11 Functional Types . . . 425
21.6.12 Union Types . . . 426
21.6.13 Utility Types . . . 426
21.6.14 Class Types . . . 427
21.6.15 Interface Types . . . 428
21.6.16 Enumeration Types . . . 428
21.6.17 Namespaces And Modules . . . 429
21.6.18 Annotations . . . 429
21.7 Verification . . . 429
21.7.1 Verification Process . . . 430
21.7.2 Dependent Entity Verification . . . 430
21.7.3 Type System Verification . . . 430
21.7.4 Control Flow Verification . . . 431
21.7.5 Abstract Interpretation . . . 431

22 Grammar Summary **433**

23 Binary Compatibility **459**
23.1 Classification of Source Code Changes . . . 460

24 Build System **463**
24.1 Package Definition . . . 463
24.2 Module Visibility . . . 464
24.3 Runtime Name Formation . . . 465
24.4 ImportPath Resolution Rules . . . 466

25 Contributors **469**

Index **471**

CHAPTER

ONE

INTRODUCTION

This document presents complete information on the new common-purpose, multi-paradigm programming language called ArkTS.

1.1 Overall Description

The ArkTS language combines and supports features that have already proven helpful and powerful in many well-known programming languages.

ArkTS supports imperative, object-oriented, functional, and generic programming paradigms, and combines them safely and consistently.

At the same time, ArkTS does not support features that allow software developers to write dangerous, unsafe, or inefficient code. In particular, the language uses the strong static typing principle. Object types are determined by their declarations, and no dynamic type change is allowed. The semantic correctness is checked at compile time.

ArkTS is designed as a part of the modern language manifold. To provide an efficient and safely executable code, the language takes flexibility and power from TypeScript and its predecessor JavaScript, and the static typing principle from Java and Kotlin. Overall design keeps the ArkTS syntax style similar to that of those languages, and some of its important constructs are almost identical to theirs on purpose.

In other words, ArkTS has a significant *common subset* of features with TypeScript, JavaScript, Java, and Kotlin. Consequently, the ArkTS style and constructs are no puzzle for the TypeScript and Java users who can intuitively sense the meaning of most constructs of the new language even if not understand them completely.

This stylistic and semantic similarity permits smoothly migrating the applications originally written in TypeScript, Java, or Kotlin to ArkTS.

Like its predecessors, ArkTS is a relatively high-level language. It means that the language provides no access to low-level machine representations. As a high-level language, ArkTS supports automatic storage management, i.e., all dynamically created objects are deallocated automatically soon after they are no longer available, and deallocating them explicitly is not required.

ArkTS is not merely a language, but rather a comprehensive software development ecosystem that facilitates the creation of software solutions in various application domains.

The ArkTS ecosystem includes the language along with its compiler, accompanying documents, guidelines, tutorials, the standard library (see *Standard Library*), and a set of additional tools that perform transition from other languages (currently, TypeScript and Java) to ArkTS automatically or semi-automatically.

The ArkTS language as a whole is characterized by the following:

- **Object orientation**

  The ArkTS language supports the *object-oriented programming* (OOP) approach based on classes and interfaces. The major notions of this approach are as follows:

  - Classes with single inheritance,
  - Interfaces as abstractions to be implemented by classes, and
  - Methods (class instance or interface methods) with overriding and dynamic dispatch mechanisms.

  Object orientation is common in many if not all modern programming languages. It enables powerful, flexible, safe, clear, and adequate software design.

- **Modularity**

  The ArkTS language supports the *modular programming* approach. It presumes that software is designed and implemented as a composition of *modules* or *top-level namespaces*.

  A *module* combines various programming resources (types, classes, functions, and so on). A module can interact with other modules by exporting all or some of its resources to, or importing from other modules.

- **Genericity**

  Some program entities in ArkTS can be *type-parameterized*. It means that an entity can represent a very high-level (abstract) concept. Providing more concrete type information constitutes the instantiation of an entity for a particular use case.

  A classical illustration is the notion of a list that represents the 'idea' of abstract data structure. An abstract notion can be turned into a concrete list by providing additional information (i.e., type of list elements).

  A similar feature (*generics* or *templates*) supported by many programming languages enables making programs and program structures more generic and reusable, and serves as a basis of the generic programming paradigm.

- **Multi-Targeting**

  ArkTS provides an efficient application development solution for a wide range of devices. The developer-friendly ArkTS ecosystem is a *cross-platform development* providing a uniform programming environment for many popular platforms. It can generate optimized applications capable of operating under the limitations of lightweight devices, or realizing the full potential of any target-specific hardware.

1.2 Lexical and Syntactic Notation

This section introduces the notation known as *context-free grammar*. The notation is used throughout this specification to define the lexical and syntactic structure of a program.

The ArkTS lexical notation defines a set of rules, or productions that specify the structure of the elementary language parts called *tokens*. All tokens are defined in *Lexical Elements*. The set of tokens (identifiers, keywords, numbers/numeric literals, operator signs, delimiters), special characters (white spaces and line separators), and comments comprises the language's *alphabet*.

The tokens defined by the lexical grammar are terminal symbols of syntactic notation. Syntactic notation defines a set of productions starting from the goal symbol *moduleDeclaration* (see *Module Declarations*). It is a sentence that consists of a single distinguished nonterminal, and describes how sequences of tokens can form syntactically correct programs.

Lexical and syntactic grammars are defined as a range of productions, and each production is comprised of the following:

- Abstract symbol (*nonterminal*) as its left-hand side,
- Sequence of one or more *nonterminal* and *terminal* symbols as its *right-hand side*,
- Character `':'` as a separator between the left- and right-hand sides, and
- Character `';'` as the end marker.

A grammar starts from the goal symbol and specifies the language, i.e., the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the left-hand-side sequence for a right-hand side of the production.

Grammars can use the following additional symbols (sometimes called *metasymbols*) in the right-hand side of a grammar production along with terminal and nonterminal symbols:

- Vertical line `'|'` to specify alternatives.
- Question mark `'?'` to specify an optional occurrence (zero- or one-time) of the preceding terminal or nonterminal.
- Asterisk `'*'` to mark a *terminal* or *nonterminal* that can occur zero or more times.
- Parentheses `'('` and `')'` to enclose any sequence of terminals and/or nonterminals marked with the metasymbols `'?'` or `'*'`.

The metasymbols specify the structuring rules for terminal and nonterminal sequences. However, they are not part of terminal symbol sequences that comprise the resultant program text.

The example below represents a production that specifies a list of expressions:

```
expressionList:
  expression (',' expression)* ','?
  ;
```

This production introduces the following structure defined by the nonterminal *expressionList*. The expression list must comprise a sequence of *expressions* separated by the terminal symbol `','`. The sequence must have at least one *expression*. The list is optionally terminated by the terminal symbol `','`.

All grammar rules are presented in *Grammar Summary* of this Specification.

1.3 Terms and Definitions

This section contains the alphabetical list of important terms found in the Specification, and their ArkTS-specific definitions. Such definitions are not generic and can differ significantly from the definitions of the same terms as used in other languages, application areas, or industries.

abstract declaration
– an ordinary interface method declaration that specifies the name and signature of a method.

array length
– a number of elements in a resizable array.

array type
– a type that consists of more than one element.

asynchronous execution
– a mode of execution where several jobs share a single worker thread (see *Execution model*).

casting conversion
– a conversion of an operand of a cast expression to an explicitly specified type.

class level scope
– a name that is declared inside a class. It is accessible inside and sometimes outside the class by means of an access modifier or a derived class.

comment
– a piece of text, insignificant for the syntactic grammar, that is added to a stream in order to document and complement source code.

compile-time error
– a text message displayed by the compiler if an error is identified in a program code that prevents the code to be generated.

compile-time warning
– a text message displayed by the compiler if a program code is found to have some logical inconsistencies that require reconsidering design and actual coding.

concurrent execution
– *asynchronous execution* or *parallel execution* (see *Execution model*).

constant
– see *constant declaration*.

constant declaration
– a declaration that introduces a new variable to which an immutable initial value can be assigned only once at the time of instantiation.

context-free grammar
– grammar in which the left-hand side of each production rule consists of only a single nonterminal symbol.

coroutine
– a suspendable job, i.e., a job with suspension points. The execution of a coroutine can be suspended and resumed later (see *Execution model*).

expression
– a formula for calculating values. The syntactic form of an expression is a combination of operators and parentheses, where the parentheses are used to change the order of calculation. The default order of calculation is determined by operator preferences.

fit into (v.)
– belong, or be implicitly convertible to an entity (see *Widening Numeric Conversions*).

fixed-size array type
– a built-in type that consists of more than one element, and has its length set only once to achieve a better performance.

function declaration
– a declaration that specifies names, signatures, and bodies when introducing a named function.

function scope
– same as *method scope*.

function type parameter scope
– a scope of a type parameter name in a function declaration. It is identical to that entire declaration.

generic
– see *generic type*.

generic type
– a named type (class or interface) that has type parameters.

goal symbol
– a sentence that consists of a single distinguished nonterminal (*moduleDeclaration*). The *goal symbol* describes how sequences of tokens can form syntactically correct programs.

grammar
– a set of rules that describe what possible sequences of terminal and nonterminal symbols a programming language interprets as correct.

Grammar is a range of productions. Each production comprises an abstract symbol (nonterminal) as its left-hand side, and a sequence of nonterminal and terminal symbols as its right-hand side. Each production contains the character `':'` as a separator between the left- and right-hand sides, and the character `';'` as the end marker.

interface level scope
– a name declared inside an interface is considered to have the interface level scope. It is accessible inside and outside the interface.

job
– a piece of code that can be executed concurrently with other jobs to communicate its return value(s) via a language-provided mechanism (see *Execution model*). The

keyword
– one of the *reserved words* that have their meanings permanently predefined in the language.

linearization
– de-nesting of all nested types of a union type to present them in the form of a flat line that includes no more union types.

literal
– a representation of a value of some type.

match (v.)
– correspond to an entity.

metasymbol
– additional symbols `'|'`, `'?'`, `'*'`, `'('`, and `')'` that can be used along with terminal and nonterminal symbols in the right-hand side of a grammar production.

method
– an ordered 3-tuple consisting of type parameters, argument types, and return types.

method scope
– a scope of a name declared immediately inside the body of a method declaration or function declaration. Method scope is identical to the body of that method declaration or function declaration from the place of declaration and up to the end of the body.

module level scope
– a name declared at the module level has a module level scope. It can be accessed within another module if it is exported from and imported into that other module.

name scope
– a region of program code within which an entity (as declared by that name) can be accessed or referred to by its simple name without any qualification.

namespace level scope
– a name declared in a namespace has a namespace level scope. It can be accessed outside the namespace if exported.

non-generic
– see *non-generic type*.

non-generic type
– a named type (class or interface) that has no type parameters.

nonterminal
– see *nonterminal symbol*.

nonterminal symbol
– a syntactically variable token that results from a successive application of production rules.

nullable type
– a variable declared to have the value `null`, or `type T | null` that can hold values of type `T` and its derived types.

nullish value
– a reference that is `null` or `undefined`.

operand
– an argument of an operation. Syntactically, operands have the form of simple or qualified identifiers that refer to variables or members of structured objects. In turn, operands can be operators with preferences ('priorities') higher than those of a given operator.

operation
– an informal notion that signifies an action or a process of operator evaluation.

operation sign
– a language token that signifies an operator and conventionally denotes a common mathematical operator, e.g., `'+'` for addition, `'/'` for division, etc. However, some languages allow using identifiers to denote operators, and/or arbitrarily combining characters that are not tokens in the alphabet of a specific language (i.e., operator signs).

operator (in programming languages)
– the term can have the following meanings:

(1) a token that denotes the action to be performed on a value (addition, subtraction, comparison, etc.).

(2) a syntactic construct that denotes an elementary calculation within an expression. An operator normally consists of an operator sign and one or more operands.

The operator sign in a unary operator with a single operand can be placed either before or after the operand (*prefix* and *postfix* unary operator respectively). An operator sign placed between two operands is called an *infix* binary operator. A conditional operator with three operands is called *ternary*.

Some operators have special notations. For example, an indexing operator is formally a binary operator which has a conventional form like a[i].

Some languages treat operators as *syntactic sugar*, i.e., a conventional version of a more common construct or *function call*. Therefore, an operator like `a+b` is conceptually handled as the call `+(a,b)`, where the operator sign acts as a function name, and the operands as function call arguments.

overloading
– a language feature that allows using a single name to call several functions (in the general sense, i.e., including methods and constructors) with different signatures and different bodies.

own (adj.)
– of a member textually declared in a class, interface, type, etc., as opposed to members inherited from a base class (superclass) or a base interface (superinterface).

parallel execution
– a mode of execution where certain jobs run on different worker threads, and hence can progress simultaneously (see *Execution model*).

production
– a sequence of terminal and nonterminal symbols that a programming language interprets as correct.

punctuator
– a token that serves to separate, complete, or otherwise organize program elements and parts, i.e., commas, semicolons, parentheses, square brackets, etc.

qualified name
– a name that consists of a sequence of identifiers separated with the token `'.'`.

resizable array type
– a built-in type that consists of more than one element, and can have the number of its constituent elements changed at runtime.

simple name
– a name that consists of a single identifier.

static member
– a class member that is not related to a particular class instance. A static member can be used across an entire program by using a qualified name notation (qualification is the name of a class).

subcomponent (derived component, child component)
– a component produced by, inherited from, and dependent from another component.

supercomponent (base component, parent component)
– a component from which another component is derived.

terminal
– see *terminal symbol*.

terminal symbol
– a syntactically invariable token (i.e., a syntactic notation defined directly by an invariable form of the lexical grammar that defines a set of productions starting from the *goal symbol*).

token
– an elementary part of a programming language, i.e., identifier, keyword, operator and punctuator, or literal. Tokens are lexical input elements that form the vocabulary of a language, and can act as terminal symbols of the syntactic grammar of a language.

tokenization
– finding the longest sequence of characters that forms a valid token (i.e., *establishing* a token) in the process of codebase reading by the machine.

type parameter scope
– the scope of a name of a type parameter that is declared in a class or an interface. Type parameter scope is identical to the entire declaration (except in static member declarations).

type reference
– references that refer to named types by specifying their type names and type arguments, where applicable, to be substituted for type parameters of the named type.

variable
– see *variable declaration*.

variable declaration
– a declaration that introduces a new named variable to which a modifiable initial value can be assigned.

white space
– lexical input elements that separate tokens from one another in order to improve the source code readability and avoid ambiguities.

widening conversion
– a conversion that causes no loss of information about the overall magnitude of a numeric value.

worker thread
– an abstraction over a platform-provided unit of parallelism, e.g., an OS thread (see *Execution model*).

CHAPTER

TWO

LEXICAL ELEMENTS

This chapter discusses the lexical structure of the ArkTS programming language.

2.1 Use of Unicode Characters

The ArkTS programming language uses characters of the Unicode Character set[1] as its terminal symbols. The Unicode UTF-16 encoding represents text in sequences of 16-bit code units.

The term *Unicode code point* is used in this specification only where such representation is relevant to refer to the Unicode Character set and the UTF-16 encoding. Where such representation is irrelevant to the discussion, the generic term *character* is used.

2.2 Lexical Input Elements

The language has the following types of *lexical input elements*:

- *White Spaces*,
- *Line Separators*,
- *Tokens*, and
- *Comments*.

2.3 White Spaces

White spaces are lexical input elements that separate tokens from one another. White spaces include the following:

- Space (U+0020),
- Horizontal tabulation (U+0009),

[1] Unicode Standard Version 15.0.0, https://www.unicode.org/versions/Unicode15.0.0/

- Vertical tabulation (U+000B),
- Form feed (U+000C),
- No-break space (U+00A0), and
- Zero-width no-break space (U+FEFF).

White spaces improve source code readability and help avoiding ambiguities. White spaces are ignored by the syntactic grammar (see *Grammar Summary*). White spaces never occur within a single token, but can occur within a comment.

2.4 Line Separators

Line separators are lexical input elements that separate tokens from one another and divide sequences of Unicode input characters into lines. Line separators include the following:

- Newline character (U+000A or ASCII <LF>),
- Carriage return character (U+000D or ASCII <CR>),
- Line separator character (U+2028 or ASCII <LS>), and
- Paragraph separator character (U+2029 or ASCII <PS>).

Line separators improve source code readability. Any sequence of line separators is considered a single separator.

Line separators are often handled as white spaces, except where line separators have special meanings. See *Semicolons* for more details.

2.5 Tokens

Tokens form the vocabulary of the language. There are four classes of tokens:

- *Identifiers*,
- *Keywords*,
- *Operators and Punctuators*, and
- *Literals*.

Token is the only lexical input element that can act as a terminal symbol of the syntactic grammar (see *Grammar Summary*). In the process of tokenization, the next token is always the longest sequence of characters that form a valid token. Tokens are separated by white spaces (see *White Spaces*), operators, or punctuators (see *Operators and Punctuators*). White spaces are ignored by the syntactic grammar (see *Grammar Summary*).

2.6 Identifiers

Identifier is a sequence of one or more valid Unicode characters. The Unicode grammar of identifiers is based on character properties specified by the Unicode Standard.

The first character in an identifier must be `'$'`, `'_'`, or any Unicode code point with the Unicode property 'ID_Start'[2]. Other characters must be Unicode code points with the Unicode property or one of the following characters:

- `'$'` (\U+0024),
- ```` ``Zero-Width Non-Joiner''' ```` (<ZWNJ>, \U+200C), or
- ```` ``Zero-Width Joiner''' ```` (<ZWJ>, \U+200D).

```
Identifier:
  IdentifierStart IdentifierPart*
  ;

IdentifierStart:
  UnicodeIDStart
  | '$'
  | '_'
  | '\\' EscapeSequence
  ;

IdentifierPart:
  UnicodeIDContinue
  | '$'
  | ZWNJ
  | ZWJ
  | '\\' EscapeSequence
  ;

ZWJ:
 '\u200C'
;
ZWNJ:
 '\u200D'
;

UnicodeIDStart
  : Letter
  | ['$']
  | '\\' UnicodeEscapeSequence;

UnicodeIDContinue
  : UnicodeIDStart
  | UnicodeDigit
  | '\u200C'
  | '\u200D';

UnicodeEscapeSequence:
  'u' HexDigit HexDigit HexDigit HexDigit
  | 'u' '{' HexDigit HexDigit+ '}'
```

(continues on next page)

[2] https://unicode.org/reports/tr31/

(continued from previous page)

```
  ;

Letter
  : UNICODE_CLASS_LU
  | UNICODE_CLASS_LL
  | UNICODE_CLASS_LT
  | UNICODE_CLASS_LM
  | UNICODE_CLASS_LO
  ;
UnicodeDigit
  : UNICODE_CLASS_ND
  ;
```

The Unicode character categories *UNICODE_CLASS_LU*, *UNICODE_CLASS_LL*, *UNICODE_CLASS_LT*, *UNICODE_CLASS_LM*, *UNICODE_CLASS_LO*, and *UNICODE_CLASS_ND* are discussed in detail in *Grammar Summary*.

2.7 Keywords

Keywords are reserved words with meanings permanently predefined in ArkTS. Keywords are case-sensitive. The exact spelling of keywords grouped by types is presented in the four tables below.

1. *Hard keywords* are reserved in any context. Hard keywords cannot be used as identifiers:

abstract	enum	let	this
as	export	native	throw
async	extends	new	true
await	false	null	try
break	final	overload	typeof
case	for	override	undefined
class	function	private	while
const	if	protected	
constructor	implements	public	
continue	import	return	
default	in	static	
do	instanceof	switch	
else	interface	super	

2. Names and aliases of predefined types are also *hard keywords*. They cannot be used as identifiers:

Primary Name	Alias
Any	
bigint	BigInt
boolean	Boolean
byte	Byte
char	Char
double	Double
float	Float
int	Int
long	Long
number	Number
Object	object
short	Short
string	String
void	

3. *Soft keywords* have special meanings reserved in certain contexts. However, soft keywords are valid identifiers elsewhere:

catch	namespace
declare	of
finally	out
from	readonly
get	set
keyof	type
module	

4. The following identifiers are also considered *soft keywords* reserved for the future use in ArkTS, or currently used as soft keywords in TypeScript:

is	memo	struct	var	yield

2.8 Operators and Punctuators

Operators are tokens that denote various actions to be performed on values. Operators include addition, subtraction, comparison, and other. The keywords `instanceof` and `typeof` also act as operators.

Punctuators are tokens that separate, complete, or otherwise organize program elements and parts. Punctuators include commas, semicolons, parentheses, square brackets, etc.

Operators and punctuators are represented by the following sequences of characters:

+	&	+=	|=	&=	<	?.
-	|	-=	^=	&&	>	!
*	^	*=	<<=	||	===	<=
/	>>	/=	>>=	++	==	>=
%	<<	%=	>>>=	--	=	...
(	)	[	]	{	}	??
,	;	.	:	!=	!==	**
**=	&&=	||=	??=			

2.9 Literals

Literals are values of certain types (see *Predefined Types* and *Literal Types*).

```
Literal:
  IntegerLiteral
  | FloatLiteral
  | BigIntLiteral
  | BooleanLiteral
  | StringLiteral
  | MultilineStringLiteral
  | NullLiteral
  | UndefinedLiteral
  | CharLiteral
  ;
```

Each literal is described in detail below. The experimental `char literal` is discussed in *char Literals*.

2.9.1 Numeric Literals

Numeric literals include integer and floating-point literals.

2.9.2 Integer Literals

Integer literals represent numbers that have neither a decimal point nor an exponential part. Integer literals can be written with radices 16 (hexadecimal), 10 (decimal), 8 (octal), and 2 (binary) as follows:

```
IntegerLiteral:
  DecimalIntegerLiteral
  | HexIntegerLiteral
```

(continues on next page)

(continued from previous page)

```
  | OctalIntegerLiteral
  | BinaryIntegerLiteral
  ;

DecimalIntegerLiteral:
  '0'
  | DecimalDigitNotZero ('_'? DecimalDigit)*
  ;

DecimalDigit:
  [0-9]
  ;

DecimalDigitNotZero:
  [1-9]
  ;

HexIntegerLiteral:
  '0' [xX]  ( HexDigit
  | HexDigit (HexDigit | '_')* HexDigit
  )
  ;

HexDigit:
  [0-9a-fA-F]
  ;

OctalIntegerLiteral:
  '0' [oO] ( OctalDigit
  | OctalDigit (OctalDigit | '_')* OctalDigit )
  ;

OctalDigit:
  [0-7]
  ;

BinaryIntegerLiteral:
  '0' [bB] ( BinaryDigit
  | BinaryDigit (BinaryDigit | '_')* BinaryDigit )
  ;

BinaryDigit:
  [0-1]
  ;
```

Integral literals with different radices are represented by the examples below:

```
153 // decimal literal
1_153 // decimal literal
0xBAD3 // hex literal
0xBAD_3 // hex literal
0o777 // octal literal
```

(continues on next page)

(continued from previous page)

```
0b101 // binary literal
```

The underscore character '_' between successive digits can be used to improve readability. Underscore characters in such positions do not change the values of literals. However, the underscore character must be neither the very first nor the very last symbol of an integer literal.

If context allows inferring type, then *Type Inference for Constant Expressions* is used to determine the type of an integer literal. Otherwise, the type is determined as follows:

- If the literal value can be represented by a non-negative 32-bit number, i.e., the value is in the range `0..max(int)`, then the type is `int`;
- If the literal value can be represented by a non-negative 64-bit number, i.e., the value is in the range `0..max(long)`, then the type is `long`;
- Otherwise, a compile-time error occurs.

The concept is represented by the examples below:

```
// literals of type int:
0
1
0x7F
0x7FFF_FFFF // max(int)

// literals of type long:
0x8000_0000
0x7FFF_FFFF_1
9223372036854775807 // max(long)

// Compile-time error, the value is too large:
9223372036854775808 // max(long) + 1
0xFFFF_FFFF_FFFF_FFFF_0
```

Note

For better compatibility with TypeScript, an integer literal cannot be used to define a negative value. Several corner cases are represented in the following example:

```
const max_int1 = 0x7FFFFFFF  // OK, type: int, value: max(int)
const max_int2 = 21474836477  // the same

const x1 = 0x80000000 // OK, type: long (!), value: 2147483648
const x2 = 2147483648  // Same

const err1: int = 2147483648 // Compile-time error, the value is out of range for 'int
↪'

const min_int = - 21474836477 - 1 // OK, type: int, value: min(int)

const max_long1 = 0x7FFF_FFFF_FFFF_FFFF // OK, type: long, value: max(long)
const max_long2 = 9223372036854775807   // Same (decimal literal)

const err2 = 0x8000_0000_0000_0000 // Compile-time error, the value is too large
const err3 = 9223372036854775808   // Compile-time error, the value is too large
```

```
16
17  // integer negation cannot be applied to a value that is too large:
18  const err4 = -9223372036854775808  // Compile-time error, the value is too large
19
20  const min_long = - max_long - 1  // OK, type: long, value: min(long)
```

2.9.3 Floating-Point Literals

Floating-point literals represent decimal numbers and consist of a whole-number part, a decimal point, a fraction part, an exponent, and a `float` type suffix as follows:

```
FloatLiteral:
    DecimalIntegerLiteral '.' FractionalPart? ExponentPart? FloatTypeSuffix?
    | '.' FractionalPart ExponentPart? FloatTypeSuffix?
    | DecimalIntegerLiteral ExponentPart? FloatTypeSuffix
    ;

ExponentPart:
    [eE] [+-]? DecimalIntegerLiteral
    ;

FractionalPart:
    DecimalDigit
    | DecimalDigit (DecimalDigit | '_')* DecimalDigit
    ;
FloatTypeSuffix:
    'f'
    ;
```

The concept is represented by the examples below:

```
1  3.14
2  3.14f
3  3.141_592
4  .5
5  1234f
6  1e10
7  1e10f
```

The underscore character `'_'` between successive digits can be used to improve readability. Underscore characters in such positions do not change the values of literals. However, the underscore character must be neither the very first nor the very last symbol of a literal.

Floating-point literals are of floating-point types that match literals as follows:

- If *float type suffix* is present, then `float`;
- Otherwise, `double` (type `number` is an alias to `double`).

If a floating-point literal is too large for its type, then a compile-time error occurs:

```
// Compile-time error, the value is too large for type float:
3.4e39f

// Compile-time error, the value is too large for type double:
1.7e309
```

2.9.4 Bigint Literals

Bigint literals represent integer numbers with an unlimited number of digits.

Bigint literals are always of type `bigint` (see *Type bigint*).

A `bigint` literal is an *integer literal* followed by the symbol `'n'`:

```
BigIntLiteral: DecimalIntegerLiteral 'n';
```

The concept is represented by the examples below:

```
153n // bigint literal
1_153n // bigint literal
-153n // negative bigint literal
```

The underscore character `'_'` between successive digits can be used to improve readability. Underscore characters in such positions do not change the values of literals. However, the underscore character must be neither the very first nor the very last symbol of a `bigint` literal.

Strings that represent numbers or any integer value can be converted to `bigint` by using built-in functions as follows:

```
BigInt(other: string): bigint
BigInt(other: long): bigint
```

Two methods allow taking *bitsCount* lower bits of a `bigint` number and return them as results. Signed and unsigned versions are both possible as follows:

```
asIntN(bitsCount: long, bigIntToCut: bigint): bigint
asUintN(bitsCount: long, bigIntToCut: bigint): bigint
```

2.9.5 Boolean Literals

Boolean literal values are represented by the two keywords: `true` and `false`.

```
BooleanLiteral:
    'true' | 'false'
    ;
```

Boolean literals are of the `boolean` type.

2.9.6 String Literals

String literals comprise zero or more characters enclosed between single or double quotes. A *multiline string* literal (see *Multiline String Literal*) is a special form of a string literal.

String literals are of the literal type that corresponds to the literal. If an operator is applied to the literal, then the literal type is replaced for `string` (see *Type string*).

```
StringLiteral:
    '"' DoubleQuoteCharacter* '"'
    | '\'' SingleQuoteCharacter* '\''
    ;

DoubleQuoteCharacter:
    ~["\\\r\n]
    | '\\' EscapeSequence
    ;

SingleQuoteCharacter:
    ~['\\\r\n]
    | '\\' EscapeSequence
    ;

EscapeSequence:
    ['"bfnrtv0\\]
    | 'x' HexDigit HexDigit
    | 'u' HexDigit HexDigit HexDigit HexDigit
    | 'u' '{' HexDigit+ '}'
    | ~[1-9xu\r\n]
    ;
```

Characters in *string literals* normally represent themselves. However, certain non-graphic characters can be represented by explicit specifications or Unicode codes. Such constructs are called *escape sequences*.

Escape sequences can represent graphic characters within a *string literal*, e.g., single quotes (`'`), double quotes (`"`), backslashes (\), and more. An escape sequence always starts with the backslash character `'\'`, followed by one of the following characters:

- `"` (double quote, U+0022),
- `'` (neutral single quote, U+0027),
- `b` (backspace, U+0008),
- `f` (form feed, U+000c),
- `n` (linefeed, U+000a),
- `r` (carriage return, U+000d),
- `t` (horizontal tab, U+0009),
- `v` (vertical tab, U+000b),
- \ (backslash, U+005c),
- `x` and two hexadecimal digits (like `7F`),

- `u` and four hexadecimal digits that form a fixed Unicode escape sequence like `\u005c`,
- `u{` and at least one hexadecimal digit followed by `}` that form a bounded Unicode escape sequence like `\u{5c}`, and
- Any single character except digits from '1' to '9' and characters `'x'`, `'u'`, `'CR'`, and `'LF'`.

Escape sequences are represented in the examples below:

```
let s1 = 'Hello, world!'
let s2 = "Hello, world!"
let s3 = "\\"
let s4 = ""
let s5 = "don't worry, be happy"
let s6 = 'don\'t worry, be happy'
let s7 = 'don\u0027t worry, be happy'
```

2.9.7 Multiline String Literal

Multiline strings can contain arbitrary text enclosed between backtick characters `` ` ``. The backlash character `'\'` is an escape for the next character. Multiline strings can contain any character except an unescaped backtick. The end of a line is handled as a newline character:

```
MultilineStringLiteral:
    '`' (BackticksContentCharacter)* '`'
    ;

BacktricksContentCharacter:
    ~[`\\\r\n]
    | '\\' EscapeSequence
    | LineContinuation
    ;

 LineContinuation:
    '\\' [\r\n\u2028\u2029]+
    ;
```

The grammar of *embeddedExpression* is described in *String Interpolation Expressions*.

A *multiline string* is represented in the example below:

```
let sentence1 =`This is an example of
               a \`multiline\` string
               to be enclosed in
               backticks`

let sentence2 = `This is an example of
a \`multiline\` string
to be enclosed in
backticks`

```

(continues on next page)

(continued from previous page)

```
11 console.log(sentence1)
12 console.log(sentence2)
```

> **Note**
>
> Leading spaces are neither squeezed nor trimmed.

The output is represented in the example below:

```
This is an example of
                a `multiline` string
                to be enclosed in
                backticks

This is an example of
a `multiline` string
to be enclosed in
backticks
```

MultilineString literals are of literal types corresponding to literals. If an operator is applied to a literal, then the literal type is replaced for `string` (see *Type string*).

2.9.8 `Undefined` Literal

Undefined literal is the only literal of types `void` and `undefined` (see *Type void or undefined*) to denote a reference with a value that is not defined. The *undefined literal* is represented by the keyword `undefined`:

```
UndefinedLiteral:
    'undefined'
    ;
```

2.9.9 `Null` Literal

Null literal is the only literal of type `null` (see *Type null*) to denote a reference without pointing at any entity. The null literal is represented by the keyword `null`:

```
NullLiteral:
    'null'
    ;
```

The value is typically used for types like `T | null` (see *Nullish Types*).

2.10 Comments

Comment is a piece of text stream added to document and complement the source code. Comments are insignificant for the syntactic grammar (see *Grammar Summary*).

Line comments begin with the sequence of characters `'//'` as in the example below, and end with the line separator character. Any character or sequence of characters between those characters is allowed but ignored.

```
// This is a line comment
```

Multiline comments begin with the sequence of characters `'\*'` as in the example below, and end with the first subsequent sequence of characters `'*/'`. Any character or sequence of characters between them is allowed but ignored.

```
/*
    This is a multiline comment
*/
```

Comments cannot be nested.

2.11 Semicolons

Declarations and statements are usually terminated by a line separator (see *Line Separators*). A semicolon must be used in some cases to separate syntax productions written in a single line, or to avoid ambiguity.

```
function foo(x: number): number {
    x++;
    x *= x;
    return x
}

let i = 1
i-i++ // one expression
i;-i++ // two expressions
```

CHAPTER

THREE

TYPES

This chapter introduces the notion of type that is one of the fundamental concepts of ArkTS and other programming languages. Type classification as accepted in ArkTS is discussed below along with all aspects of using types in programs written in the language.

The type of an entity is conventionally defined as the set of *values* the entity (variable) can take, and the set of *operators* applicable to the entity of a given type.

ArkTS is a statically typed language. It means that the type of every declared entity and every expression is known at compile time. The type of an entity is either set explicitly by a developer, or inferred implicitly (see *Type Inference*) by the compiler.

The types integral to ArkTS are called *predefined types* (see *Predefined Types*).

The types introduced, declared, and defined by a developer are called *user-defined types*. All *user-defined types* must have complete type declarations presented as source code in ArkTS.

ArkTS types are summarized in the table below:

Predefined Types	User-Defined Types
`byte`, `short`,	class types,
`int`, `long`,	interface types,
`float`, `double`,	tuple types,
`number`,	union types,
`boolean`, `char`,	function types,
`string`,	enumeration types,
`bigint`,	type parameters,
`Any`, `Object`,	string literal types
`never`, `void`,	
`undefined`, `null`,	
`Array<T>` or `T[]`,	
`FixedArray<T>`,	
`ValueArray<T>`	

Note

Type `number` is an alias to `double`.

Most *predefined types* have aliases to improve TypeScript compatibility as follows:

Primary Name	Alias
number	Number
byte	Byte
short	Short
int	Int
long	Long
float	Float
double	Double
boolean	Boolean
char	Char
string	String
bigint	BigInt
Object	object

Using primary names of *predefined types* is recommended in all cases.

3.1 Predefined Types

Predefined types include the following:

- *Value Types*;
- *Type Any*;
- *Type Object*;
- *Type never*;
- *Type void or undefined*;
- *Type null*;
- *Type string*;
- *Type bigint*;
- *Array Types* (`Array<T>` or `T[]` or `FixedArray<T>`);
- *Type Function*.

3.2 User-Defined Types

User-defined types include the following:

- Class types (see *Classes*);
- Interface types (see *Interfaces*);
- Enumeration types (see *Enumerations*);

- *Function Types*;
- *Tuple Types*;
- *Union Types*;
- *Type Parameters*; and
- *Literal Types*.

3.3 Using Types

Source code can refer to a type by using the following:

- Type reference for:
  - *Named Types*, or
  - Type aliases (see *Type Alias Declaration*);
- In-place type declaration for:
  - *Array Types*,
  - *Tuple Types*,
  - *Function Types*,
  - *Function Types with Receiver*,
  - *Keyof Types*,
  - *Union Types*, or
  - Type in parentheses.

The syntax of *type* is presented below:

```
type:
    annotationUsageWithParentheses?
    ( typeReference
    | 'readonly'? arrayType
    | 'readonly'? tupleType
    | functionType
    | functionTypeWithReceiver
    | unionType
    | keyofType
    | StringLiteral
    )
    | '(' type ')'
    ;
```

The usage of annotations is discussed in *Using Annotations*.

Types with the prefix `readonly` are discussed in *Readonly Array Types* and *Readonly Tuple Types*.

The usage of types is represented by the example below:

```
let n: number   // Using identifier as a predefined value type name
let o: Object   // Using identifier as a predefined class type name
let a: number[] // Using array type
let t: [number, number] // Using tuple type
let f: ()=>number      // Using function type
let u: number|string    // Using union type
let l: "xyz"            // Using string literal type

class C { n = 1; s = "aa"}
let k: keyof C  // Using keyof to build union type
```

Parentheses are used to specify the required type structure if the type is a combination of array, function, or union types. Without parentheses, the symbol '|' that constructs a union type has the lowest precedence as represented by the example below:

```
// A nullish array with elements of type string:
let a: string[] |  undefined
let s: string[] = []

a = s    // OK
a = undefined // OK, a is nullish type

// An array with elements whose types are string or undefined:
let b1: (string | undefined)[]
b1 = undefined // Error, b1 is an array and is not nullish type
b1 = ["aa", undefined] // OK

// string or array of undefined elements:
let b2: string | undefined[]
b2 = undefined // Error, b2 - string or array of undefined - not nullish
b2 = [undefined, undefined] // OK

// A function type that returns string or undefined
let c: () => string | undefined
c = undefined // error, c is not nullish
c = (): string | undefined => { return undefined } // OK

// (A function type that returns string) or undefined
let d: (() => string) | undefined
d = undefined // OK, d is nullish
d = (): string => { return "hi" } // OK
```

The annotation which is used in front of type always needs parentheses (as highlighted in grammar snippet above). The reason is to prevent ambiguity between annotation parentheses and parentheses used in a type declaration:

```
let var_name1: @my_annotation() (A|B) // OK
let var_name2: @my_annotation (A|B)  // Compile-time error
```

3.4 Named Types

Named types are classes, interfaces, enumerations, aliases, type parameters, and predefined types (see *Predefined Types*), except built-in arrays. Other types (i.e., array, function, and union types) are anonymous unless aliased. Respective named types are introduced by the following:

- Class declarations (see *Classes*),
- Interface declarations (see *Interfaces*),
- Enumeration declarations (see *Enumerations*),
- Type alias declarations (see *Type Alias Declaration*), and
- Type parameter declarations (see *Type Parameters*).

Classes, interfaces and type aliases with type parameters are *generic types* (see *Generics*). Named types without type parameters are *non-generic types*.

Type references (see *Type References*) refer to named types by specifying their type names and (where applicable) type arguments to be substituted for the type parameters of a named type.

3.5 Type References

Type reference refers to a type by one of the following:

- *Simple* or *qualified* type name (see *Names*),
- Type alias (see *Type Alias Declaration*).

Type reference that refers to a generic class or to an interface type is valid if it is a valid instantiation of a generic. Its type arguments can be provided explicitly or implicitly based on defaults.

The syntax of *type reference* is presented below:

```
typeReference:
    qualifiedName typeArguments?
    ;
```

```
let map: Map<string, number> // Map<string, number> is the type reference

class A<T> {...}
class C<T> {
   field1: A<T>  // A<T> is a class type reference - class type reference
   field2: A<number> // A<number> is a type reference - class type reference
   foo (p: T) {} // T is a type reference - type parameter
   constructor () { /* some body to init fields */ }
}

type MyType<T> = A<T>[]
let x: MyType<number> = [new A<number>, new A<number>]
  // MyType<number> is a type reference  - alias reference
  // A<number> is a type reference - class type reference
```

If *type reference* refers to a type by a type alias (see *Type Alias Declaration*), then the type alias is replaced for a non-aliased type in all cases when dealing with types. The replacement is potentially recursive.

```
type T1 = Object
type T2 = number
function foo(t1: T1, t2: T2)  {
    t1 = t2      // Type compatibility test will use Object and number
    t2 = t2 + t2 // Operator validity test will use type number not T2
}
```

3.6 Value Types

Value types are predefined integer types (see *Integer Types and Operations*), floating-point types (see *Floating-Point Types and Operations*), the boolean type (see *Type boolean*), and the character type (see *Type char*).

The values of such types do *not* share state with other values.

Note

Tables in *Unary Expressions* and *Binary Expressions* summarize information about valid combinations of operand types and resultant types of unary and binary operations.

3.6.1 Numeric Types

Numeric types are integer and floating-point types (see *Integer Types and Operations* and *Floating-Point Types and Operations*).

Each numeric type has its own set of values. *Widening Numeric Conversions* are used in expressions to convert a value of a smaller type to a value of a larger type.

Numeric type hierarchy is defined in the following way:

- `byte` < `short` < `int` < `long` < `float` < `double`, where the smallest type is `byte`, and the largest type is `double`.

Type `bigint` does not belong to this hierarchy. No implicit conversion from numeric types to `bigint` occurs. The methods of class `BigInt` (which is a part of *Standard Library*) must be used to create `bigint` values from numeric type values.

Numeric types are represented by classes of the *Standard Library*. It means that *numeric types* are of the class type nature, that all of them are subtypes of `Object`, and therefore can be used at any place where a class name is expected.

```
let a_number = new number
let a_byte = new byte
let an_integer = new int
console.log (a_number, a_byte, an_integer)
// Output is: 0 0 0
```

3.6.2 Integer Types and Operations

Type	Corresponding Set of Values
`byte`	All signed 8-bit integers (-2^7 to $2^7 - 1$)
`short`	All signed 16-bit integers (-2^{15} to $2^{15} - 1$)
`int`	All signed 32-bit integers (-2^{31} to $2^{31} - 1$)
`long`	All signed 64-bit integers (-2^{63} to $2^{63} - 1$)
`bigint`	All integers with no limits

Note

Type bigint is not a numeric type, yet it operates with integer values of arbitrary precision. That is why it is discussed in detail in the appropriate subsection.

ArkTS provides a number of operators to act on integer values as discussed below.

- Comparison operators that produce a value of type `boolean`:
  - Numeric relational operators `'<'`, `'<='`, `'>'`, and `'>='` (see *Numeric Relational Operators*);
  - Numeric equality operators `'=='` and `'!='` (see *Numeric Equality Operators*);
- Exponentiation operator `'**'` as a variant that produces a value of type `bigint` (see *Exponentiation Expression*);
- Numeric operators that produce values of types `int`, `long`, or `bigint`:
  - Unary plus `'+'` and minus `'-'` operators (see *Unary Plus* and *Unary Minus*);
  - Multiplicative operators `'*'`, `'/'`, and `'%'` (see *Multiplicative Expressions*);
  - Additive operators `'+'` and `'-'` (see *Additive Expressions*);
  - Increment operator `'++'` used as prefix (see *Prefix Increment*) or postfix (see *Postfix Increment*);
  - Decrement operator `'--'` used as prefix (see *Prefix Decrement*) or postfix (see *Postfix Decrement*);
  - Signed and unsigned shift operators `'<<'`, `'>>'`, and `'>>>'` (see *Shift Expressions*);
  - Bitwise complement operator `'~'` (see *Bitwise Complement*);
  - Integer bitwise operators `'&'`, `'^'`, and `'|'` (see *Integer Bitwise Operators*);
- Ternary conditional operator `'?:'` (see *Ternary Conditional Expressions*);
- String concatenation operator `'+'` (see *String Concatenation*) that, if one operand is `string` and the other is of an integer type, converts the integer operand to `string` with the decimal form, and then creates a concatenation of the two strings as a new `string`.

If either operand of a binary integer operation except *Shift Expressions* is of type `long` and the other operand is of a lesser type, then numeric conversion (see *Widening Numeric Conversions*) is used to widen the second operand first to type `long`. In this case:

- Operation implementation uses 64-bit precision; and
- Result of the numeric operator is of type `long`.

If otherwise neither operand is of type `long` and any operand is of a type other than `int`, then numeric conversion is used to widen the latter first to type `int`. In this case:

- Operation implementation uses the 32-bit precision; and
- Result of the numeric operator is of type `int`.

Conversions between integer types and type `boolean` are not allowed. However, the value of integer type can be used as a logical condition in some cases (see *Extended Conditional Expressions*)

The integer operators cannot indicate an overflow or an underflow.

An integer operator can throw `ArithmeticError` if the right-hand-side operand of an integer division operator '`/`' (see *Division*) and an integer remainder operator `'%'` (see *Remainder*) is zero. The situation is discussed in *Error Handling*.

Predefined constructors, methods, and constants for *integer types* are parts of the ArkTS *Standard Library*.

3.6.3 Floating-Point Types and Operations

Type	Corresponding Set of Values
`float`	The set of all IEEE 754[3] 32-bit floating-point numbers
`number`, `double`	The set of all IEEE 754 64-bit floating-point numbers

ArkTS provides a number of operators to act on floating-point type values as discussed below.

- Comparison operators that produce a value of type *boolean*:
  - Numeric relational operators `'<'`, `'<='`, `'>'`, and `'>='` (see *Numeric Relational Operators*);
  - Numeric equality operators `'=='` and `'!='` (see *Numeric Equality Operators*);
- Numeric operators that produce values of type `float` or `double`:
  - Unary plus `'+'` and minus `'-'` operators (see *Unary Plus* and *Unary Minus*);
-
  - Exponentiation operator `'**'` as a variant that produces a value of type `double` (see *Exponentiation Expression*);
  - Multiplicative operators `'*'`, `'/'`, and `'%'` (see *Multiplicative Expressions*);
  - Additive operators `'+'` and `'-'` (see *Additive Expressions*);
  - Increment operator `'++'` used as prefix (see *Prefix Increment*) or postfix (see *Postfix Increment*);
  - Decrement operator `'--'` used as prefix (see *Prefix Decrement*) or postfix (see *Postfix Decrement*);
- Numeric operators that produce values of type `int` or `long`:
  - Signed and unsigned shift operators `'<<'`, `'>>'`, and `'>>>'` (see *Shift Expressions*);
  - Bitwise complement operator `'~'` (see *Bitwise Complement*);
  - Integer bitwise operators `'&'`, `'^'`, and `'|'` (see *Integer Bitwise Operators*);
- Ternary conditional operator `'?:'` (see *Ternary Conditional Expressions*);

[3] Any mention of IEEE 754 in this Specification refers to the latest revision of "754-2019–IEEE Standard for Floating-Point Arithmetic".

- The string concatenation operator `'+'` (see *String Concatenation*) that, if one operand is of type `string` and the other is of a floating-point type, converts the floating-point type operand to type `string` with a value represented in the decimal form (without loss of information), and then creates a concatenation of the two strings as a new `string`.

An operation is called a *floating-point operation* when:

- Both operands of an `exponentiation operator` are of a numeric type (see *Exponentiation Expression*); or
- At least one operand in a binary operator is of a floating-point type even if the other operand is integer, and the operator is not a string concatenation.

The operation implementation for an `exponentiation operator` with numeric operands always uses the 64-bit floating-point arithmetic. Operands are converted to `double` values if needed. The result of the numeric operator is a value of type `double`.

The following rules apply to other floating-point operations:

- If at least one operand of the numeric operator is of type `double`, then the operation implementation uses the 64-bit floating-point arithmetic. The result of the numeric operator is a value of type `double`.
- If the first operand is of type `double` and the other operand is not, then the numeric conversion (see *Widening Numeric Conversions*) is used to widen the operand first to type `double`.
- If neither operand is of type `double`, then the operation implementation is to use the 32-bit floating-point arithmetic. The result of the numeric operator is a value of type `float`.
- If the first operand is of type `float` and the other operand is not, then the numeric conversion is used to widen the operator first to type `float`.

Any floating-point type value can be cast to or from any numeric type (see *Numeric Types*).

Conversions between floating-point types and type `boolean` are not allowed. However, the value of floating-point type can be used as a logical condition in some cases (see *Extended Conditional Expressions*)

Operators on floating-point numbers, except the remainder operator (see *Remainder*), behave in compliance with the IEEE 754 Standard. For example, ArkTS requires the support of IEEE 754 *denormalized* floating-point numbers and *gradual underflow* which facilitate proving the desirable properties of a particular numeric algorithm. Floating-point operations do not *flush to zero* if the calculated result is a denormalized number.

ArkTS requires the floating-point arithmetic to behave as if the floating-point result of every floating-point operator is rounded to the result precision. An *inexact* result is rounded to a representable value nearest to the infinitely precise result. ArkTS uses the *round to nearest* principle (the default rounding mode in IEEE 754), and prefers the representable value with the least significant bit zero out of any two equally near representable values.

ArkTS uses *round toward zero* to convert a floating-point value to an integer value (see *Numeric Casting Conversions*). In this case it acts as if the number is truncated, and the mantissa bits are discarded. The result of *rounding toward zero* is the value of the format that is closest to and no greater in magnitude than the infinitely precise result.

A floating-point operation with overflow produces a signed infinity.

A floating-point operation with underflow produces a denormalized value or a signed zero.

A floating-point operation with no mathematically definite result produces `NaN`.

All numeric operations with a `NaN` operand result in `NaN`.

Predefined constructors, methods, and constants for *floating-point types* are parts of the ArkTS *Standard Library*.

3.6.4 Type boolean

Type `boolean` represents logical values `true` and `false`.

The boolean operators are as follows:

- Equality operators (see *Equality Expressions*);
- Logical complement operator `'!'` (see *Logical Complement*);
- Logical operators `'&'`, `'^'`, and `'|'` (see *Boolean Logical Operators*);
- Conditional-and operator `'&&'` (see *Conditional-And Expression*) and conditional-or operator `'||'` (see *Conditional-Or Expression*);
- Ternary conditional operator `'?:'` (see *Ternary Conditional Expressions*);
- String concatenation operator `'+'` (see *String Concatenation*) that converts an operand of type `boolean` to type `string` (`true` or `false`), and then creates a concatenation of the two strings as a new `string`.

Type `boolean` is a class type that is a part of the *Standard Library*. It means that type `boolean is a subtype of ``Object`, and therefore can be used at any place where a class name is expected.

```
let a_boolean = new boolean
console.log (a_boolean)
// Output is: false
let o: Object = a_boolean // OK
```

3.7 Reference Types

Reference types can be of the following kinds:

- *Class* types (see *Type Object* and *Classes*);
- *Interface* types (see *Interfaces*);
- *Array Types*;
- *Fixed-Size Array Types*;
- *Tuple Types*;
- *Function Types*;
- *Union Types*;
- *Literal Types*;
- *Type Any*;
- *Type string*;
- *Type bigint*;
- *Type never*;
- *Type null*;
- *Type void or undefined*; and
- *Type Parameters*.

The term *Object* is used further in this document to denote any instance pointed at by a variable or constant of a reference type (see *Variable and Constant Declarations*).

Multiple references to an object are possible.

Objects can have states. A state of an object that is a class instance is stored in its fields. A state of an object that is an array (see *Array Types*) or a tuple (see *Tuple Types*) is stored in its elements.

If two variables of a reference type contain references to the same object, and either variable modifies the state of that object, then the change of state is also visible in the other variable.

3.8 Type Any

Type `Any` is a predefined type which is the supertype of all types. Type `Any` is a predefined *nullish-type* (see *Nullish Types*), i.e., a supertype of *Type void or undefined* and *Type null* in particular.

Type `Any` has no methods or fields.

3.9 Type Object

Type `Object` is a predefined class type which is the supertype (see *Subtyping*) of all types except *Type void or undefined*, *Type null*, *Nullish Types*, *Type Parameters*, and *Union Types* that contain type parameters. Type `Object` is a subtype of type `Any` (see *Type Any*). All subtypes of `Object` inherit all methods of class `Object` (see *Inheritance*). All methods of class `Object` are described in full in *Standard Library*.

The method `toString` used in the examples in this document returns a string representation of any object.

3.10 Type never

Type `never` is assignable to any type (see *Assignability*).

Type `never` has no instance. Type `never` is used as one of the following:

- Return type for functions or methods that never return a value, but throw an error when completing an operation.
- Type of variables that never get a value (however, an assignment statement with type `never` in both left-hand and right-hand sides is valid).
- Type of parameters of a function or a method to prevent the body of that function or method from being executed.

```
function foo (): never {
   // function foo never returns
   throw new Error("foo() never returns")
}

let x: never = foo() // x never gets a value

function bar (p: never) {
```

(continues on next page)

(continued from previous page)

```
    // function bar body is never executed
}

bar (foo()) // bar is never called as foo() call never returns
```

3.11 Type void or undefined

Type names `void` and `undefined` in fact refer to the same type with the single value named `undefined` (see *Undefined Literal*), and are used in this document interchangeably.

```
function f1 (): void {
    return undefined // OK
}

function f2 (): undefined {
    return // OK
}

function f3 () {
    return undefined // OK
}

let v: void = undefined
let u: undefined = undefined
v = u // OK
u = v // OK
```

Type name `void` is used typically as a return type to highlight that a function, a method, or a lambda can contain *return Statements* with no expression, or no return statement at all:

```
function foo (): void {} // no return at all

class C {
    bar(): void {
        return // with no expression
    }
}

type FunctionWithNoParametersType = () => void

let funcTypeVariable: FunctionWithNoParametersType = (): void => {}
```

Type name `void` can be used as a type argument that instantiates a generic type, function, or method as follows:

```
class A<T> {
   f: T
   m(): T { return this.f }
   constructor (f: T) { this.f = f }
```

(continues on next page)

(continued from previous page)

```
}
let a1 = new A<void>(undefined)      // OK, type name void used as type argument
let a2 = new A<undefined>(undefined) // OK, type name undefined used as type argument

console.log (a1.f, a2.m()) // Output is "undefined" "undefined"

function foo<T>(p: T): T { return p }
foo<void>(undefined) // OK, returns 'undefined' value

type F1<T> = () => T
const f1: F1<void> = (): void => {}
const f2: F1<void> = () => {}
const f3: F1<void> = (): undefined => { return undefined }

// Array literals can be assigned to the array of void type name in any form
type A1<T> = T[]
type A2<T> = Array<T>
const a1: A1<void> = [undefined]
const a2: A2<void> = [undefined, undefined]
const a3: void[] = [undefined]
```

Type name `undefined` can be used also as type argument to instantiate a generic type as follows:

```
class A<T> {}
let a = new A<undefined>() // OK, type parameter is irrelevant
function foo<T>(x: T) {}

foo<undefined>(undefined) // OK
```

3.12 Type `null`

The only value of type `null` is the literal `null` (see *Null Literal*).

Note

- Type `null` is supported for TypeScript compatibility.
- Using `undefined` instead of `null` is considered the best practice in TypeScript and JavaScript.
- Using type `null` as type annotation or in *Nullish Types* is not recommended if not required specifically. Using type `undefined` is recommended instead.
- Type `undefined` provides a better performance than type `null` does.

3.13 Type string

Type string values are all string literals, e.g., 'abc'. Type string stores sequences of characters as Unicode UTF-16 code units.

A string object is immutable, the value of a string object cannot be changed after the object is created. The value of a string object can be shared.

Type string has dual semantics, i.e.:

- Type string behaves like a reference type (see *Reference Types*) if created, assigned, or passed as an argument;
- Type string is handled as a value type by all string operations (see *String Concatenation*, *Equality Expressions*, and *String Relational Operators*).

As a result, reference type semantics of type string highlights that this type is a class type. The appropriate class is a part of the *Standard Library*, and type string is a subtype of Object, and can be used at any place where a class name is expected.

Moreover, type string is iterable (see *Iterable Types*), and can be used at any place where an iterable type is expected.

```
let a_string = new string
console.log (a_string)
// Output is: <empty_string>
let o: Object = a_string // OK
```

A number of operators can act on string values as follows:

- Accessing the property length returns string length as int type value. String length is a non-negative integer number. String length is set once at runtime and cannot be changed after that.
- Concatenation operator '+' (see *String Concatenation*) produces a value of type string. If the result is not a constant expression (see *Constant Expressions*), then the string concatenation operator can implicitly create a new string object;
- Indexing a string value (see *String Indexing Expression*) returns a value of type string. A new string object can be created implicitly.

A string value can contain any character, i.e., no character can be used to indicate the end of a string. A character with the value '0' is an ordinary character inside a string as represented by the following example:

```
console.log("a\0b".length) // output: 3
```

Using string in all cases is recommended, although the name String also refers to type string.

3.14 Type bigint

ArkTS has the built-in bigint type that allows handling theoretically arbitrary large integers. Values of type bigint can hold numbers that are larger than the maximum value of type long. Type bigint uses the arbitrary-precision arithmetic. Type bigint does not belong to the hierarchy of *Numeric Types*. The consequences are as follows:

- No implicit conversion between bigint type and numeric types.
- Relational operators that use an operand of type bigint along with an operand of another type are illegal. Attempting to use such a relational operator produces a compile-time error.

- Binary arithmetic expressions that use an operand of type `bigint` along with an operand of another type are illegal and produce a compile-time error.
- The equality expression with `bigint` against non-`bigint` always returns `false`, and causes a compile-time warning.

Values of type `bigint` can be created from the following:

- *Bigint literals* (see *Bigint Literals*); or
- Numeric type values, by using a call to the standard library class `BigInt` methods or constructors (see *Standard Library*).

Similarly to `string`, `bigint` type has dual semantics:

- If created, assigned, or passed as an argument, type `bigint` behaves like a reference type (see *Reference Types*).
- All applicable operations handle type `bigint` as a value type, meaning the values of this type do not share state with other values.

Using `bigint` is recommended in all cases, although the name `BigInt` also refers to type `bigint`. For TypeScript compatibility, objects of type `bigint` can be created with help of `BigInt` static methods.

```
let b1: bigint = new BigInt(5) // for Typescript compatibility
let b2: bigint = 123n
```

Type `bigint` is a class type that has an appropriate class as a part of the *Standard Library*. It means that type `bigint` is a subtype of `Object`, and therefore can be used at any place where a class name is expected.

```
let a_bigint = new bigint
console.log (a_bigint)
// Output is: 0
let o: Object = a_bigint // OK
```

3.15 Literal Types

Literal types are aligned with some ArkTS literals (see *Literals*). Their names are the same as the names of their values, i.e., literals proper. ArkTS supports only the following literal types:

- *String Literal Types*,
- `null`, and
- `undefined`.

```
let a: "string literal" = "string literal"
let b: null = null
let c: undefined = undefined

printThem (a, b, c)
function printThem (p1: "string literal", p2: null, p3: undefined) {
    console.log (p1, p2, p3)
}
```

There are no operations for literal types `null` and `undefined`.

3.15.1 String Literal Types

Operations on variables of string literal types are identical to the operations of their supertype `string` (see *Type string*). The resulting operation type is the type specified for the operation in the supertype:

```
let s0: "string literal" = "string literal"
let s1: string = s0 + s0   // + for string returns string
```

3.16 Array Types

Array type represents a data structure intended to comprise any non-negative number of elements of types that are subtypes of the type specified in the array declaration. ArkTS supports the following two predefined array types:

- *Resizable Array Types*; and
- *Fixed-Size Array Types* as an experimental feature.

Resizable array types are recommended for most cases. *Fixed-size array types* can be used where performance is the major requirement.

Fixed-size arrays differ from *resizable arrays* as follows:

- *Fixed-size arrays* have their length set only once to achieve a better performance.
- *Fixed-Size arrays* have no methods defined.

Any array type is a class type that has an appropriate class in the *Standard Library*. It means that array types are subtypes of `Object`, and that they can be used at any place where a class name is expected. Moreover, array types are iterable (see *Iterable Types*), and can be used at any place where an iterable type is expected.

Note

The term *array type* as used in this Specification applies to both *resizable array type* and *fixed-size array type*. The same holds true for *array value* and *array instance*. *Resizable arrays* and *fixed-size arrays* are not assignable to each other.

3.16.1 Resizable Array Types

Resizable array type is a built-in type characterized by the following:

- Any object of resizable array type contains elements. The number of elements is known as *array length*, and can be accessed by using the property `length`.
- Array length is a non-negative integer number.
- Array length can be set and changed at runtime.

- Array element is accessed by its index. The index is an integer number in the range from *0* to *array length minus 1*.
- Accessing an element by its index is a constant-time operation.
- If passed to non-ArkTS environment, an array is represented as a contiguous memory location.
- Type of each array element is assignable to the element type specified in the array declaration (see *Assignability*).

Resizable array type with elements of type `T` can have the following two forms of syntax:

- `T[]`, and
- `Array<T>`.

The first form uses the following syntax:

```
arrayType:
   type '[' ']'
   ;
```

Note

`T[]` and `Array<T>` specify identical, i.e., indistinguishable types (see *Type Identity*).

Two basic operations with array elements take elements out of, and put elements into an array by using the operator '`[]`'.

The same syntax can be used to work with *Indexable Types*, some of such types are parts of *Standard Library*.

The number of elements in an array can be obtained by accessing the property `length`. The length of an array can be set and changed in runtime using the methods defined in *Standard Library*.

An array can be created by using *Array Literal*, *Resizable Array Creation Expressions*, or the constructors defined in *Standard Library*.

ArkTS allows setting a new value to `length` to shrink an array and provide better TypeScript compatibility. An error is caused by the following situations:

- The value is of type `number` or other floating-point type, and the fractional part differs from 0;
- The value is less than zero; or
- The value is greater than previous length.

The above situations cause errors as follows:

- A runtime error, if the situation is identified at runtime, i.e., during program execution; and
- A compile-time error, if the situation is detected during compilation.

Array operations are illustrated below:

```
let a : number[] = [0, 0, 0, 0, 0]
  /* allocate array with 5 elements of type number */
a[1] = 7 /* put 7 as the 2nd element of the array, index of this element is 1 */
let y = a[4] /* get the last element of array 'a' */
let count = a.length // get the number of array elements
a.length = 3 // shrink array
y = a[2] // OK, 2 is the index of the last element now
y = a[3] // Runtime error, attempt to access a non-existing array element
```

(continues on next page)

(continued from previous page)

```

let b: Array<number> = a // 'b' points to the same array as 'a'
```

A type alias can set a name for an array type (see *Type Alias Declaration*):

```
type Matrix = number[][] /* array of array of numbers */
```

An array as an object is assignable to a variable of type `Object`:

```
let a: number[] = [1, 2, 3]
let o: Object = a
```

3.16.2 Readonly Array Types

Readonly array type is immutable, i.e.:

- Length of a variable of a *readonly array type* cannot be changed;
- Elements of a *readonly array type* cannot be modified after the initial assignment directly nor through a function or method call.

Otherwise, a compile-time error occurs.

```
let x: readonly number [] = [1, 2, 3]
x[0] = 42 // Compile-time error, array itself is readonly
```

Readonly array type with elements of type `T` can have the following two syntax forms:

- `readonly T[]`, and
- `ReadonlyArray<T>`.

Both forms specify identical (indistinguishable) types (see *Type Identity*).

> **Note**
>
> In arrays of arrays, all arrays are `readonly`.

3.17 Tuple Types

Tuple type is a reference type created as a fixed set of other types.

The syntax of *tuple type* is presented below:

```
tupleType:
    '[' (type (',' type)* ','?)? ']'
    ;
```

The value of a tuple type is a group of values of types that comprise the tuple type. The number of values in the group equals the number of types in a tuple type declaration. The order of types in a tuple type declaration specifies the type of the corresponding value in the group.

It implies that each element of a tuple has its own type. The operator `'[]'` (square brackets) is used to access the elements of a tuple in a manner similar to accessing the elements of an array.

An index expression operand must be a *constant expression* (see *Constant Expressions*) of an integer type. The index of the first tuple element is *0*:

```
let tuple: [number, number, string, boolean, Object] =
           [     6,      7,  "abc",    true,    42]
tuple[0] = 42
console.log (tuple[0], tuple[4]) // `42 42` be printed
```

The number of elements of a tuple is known as *tuple length*, and can be accessed by using the property `length`:

```
let tuple: [number, string]  = [1, "" ]
console.log(tuple.length) // output: 2
```

Using the property `length` make sense for *Type Tuple*.

Any tuple type is subtype of type `Tuple` (see *Type Tuple*). Subtyping for tuples is discussed in *Subtyping for Tuple Types*.

3.17.1 Readonly Tuple Types

If an *tuple* type has the prefix `readonly`, then its elements cannot be modified after the initial assignment directly or through a function or method call. Otherwise, a compile-time error occurs as follows:

```
let x: readonly [number, string] = [1, "abc"]
x[0] = 42 // Compile-time error, tuple itself is readonly
```

3.17.2 Type `Tuple`

Type `Tuple` is a predefined type that is an *abstract superclass* of any tuple type.

```
let pair: [number, string] = [1, "abc"]

let a: Tuple = pair // OK, subtyping
```

An empty tuple type is identical to `Tuple`:

```
let empty: [] = [] // empty tuple with no elements in it
```

Type `Tuple` is preserved by *Type Erasure*, and can be used in *instanceof Expression* and *Cast Expression*.

An element of a `Tuple` value cannot be accessed directly. The method `unsafeGet` with the signature can be used instead to get an element value:

```
unsafeGet(index: int): Any
```

Calls of the method `unsafeGet` cause a runtime error if:

- `index` value is less than zero; or
- `index` value is greater than or equal to the actual tuple length.

The usage of the method `unsafeGet` is illustrated below:

```
function log_1(x: Object) {
    if (x instanceof Tuple) {
        console.log(x.unsafeGet(1))
    }
}

let a: [string, string] = ["aa", "bb"]
log_1(a) // OK, output: 2, "bb"

let b: [string] = ["aa"]
log_1(b)     // Runtime error in ``unsafeGet`` call
```

No element of a `Tuple` value can be changed. ArkTS supports no such change as it can cause a runtime error at an unpredictable place during execution.

3.18 Function Types

Function type can be used to express the expected signature of a function. A function type consists of the following:

- Optional type parameters;
- List of parameters (which can be empty);
- Optional return type.

The syntax of *function type* is as follows:

```
functionType:
    '(' ftParameterList? ')' ftReturnType
    ;

ftParameterList:
    ftParameter (',' ftParameter)* (',' ftRestParameter)?
    | ftRestParameter
    ;

ftParameter:
    identifier ('?')? ':' type
    ;

ftRestParameter:
    '...' ftParameter
```

(continues on next page)

(continued from previous page)

```
    ;

ftReturnType:
    '=>' type
    ;
```

The `rest` parameter is described in *Rest Parameter*.

```
let binaryOp: (x: number, y: number) => number
function evaluate(f: (x: number, y: number) => number) { }
```

A type alias can set a name for a *function type* (see *Type Alias Declaration*):

```
type BinaryOp = (x: number, y: number) => number
let op: BinaryOp
```

If a function type has the `'?'` mark for a parameter name, then this parameter and all parameters that follow (if any) are optional. Otherwise, a compile-time error occurs. The actual type of the parameter is then a union of the parameter type and type `undefined`. This parameter has no default value.

```
type FuncTypeWithOptionalParameters = (x?: number, y?: string) => void
let foo: FuncTypeWithOptionalParameters
    = ():void => {}          // OK, arguments are just ignored
foo = (p: number):void => {} // Compile-time error, call with zero arguments is invalid
foo = (p?: number):void => {} // OK, call with zero or one argument is valid
foo = (p1: number, p2?: string):void => {} // Compile-time error, call with zero␣
↪arguments is invalid
foo = (p1?: number, p2?: string):void => {} // OK

foo()
foo(undefined)
foo(undefined, undefined)
foo(42)
foo(42, undefined)
foo(42, "a string")

type IncorrectFuncTypeWithOptionalParameters = (x?: number, y: string) => void
   // Compile-time error, no mandatory parameter can follow an optional parameter

function bar (
   p1?: number,
   p2:  number|undefined
) {
   p1 = p2 // OK
   p2 = p1 // OK
   // Types of p1 and p2 are identical
}
```

More details on function types assignability are provided in *Subtyping for Function Types*.

3.18.1 Type Function

Type Function is a predefined type that is a *direct superinterface* of any function type.

A value of type Function cannot be called directly. A developer must use the unsafeCall method instead. This method checks the arguments of type Function, and calls the underlying function value if the number and types of the arguments are valid.

```
function foo(n: number) {}

let f: Function = foo

f(1) // Compile-time error, cannot be called

f.unsafeCall(3.14) // correct call and execution
f.unsafeCall() // Runtime error, wrong number of arguments
```

Another important property of type Function is name. It is a string that contains the name associated with the function object in the following way:

- If a function or a method is assigned to a function object, then the associated name is that of the function or of the method;
- If a lambda is assigned to a variable of Function type, then the associated name is that of the variable;
- Otherwise, the string is empty.

```
function print_name (f: Function) {
   console.log (f.name)
}

function foo() {}
print_name (foo) // Output: "foo"

class A {
   static sm() {}
   m() {}
}
print_name (A.sm)      // Output: "sm"
print_name (new A().m) // Output: "m"

let x: Function = (): void => {}
print_name (x) // Output: "x"

let y = x
print_name (y) // Output: "x"

print_name ():void=>{}) // Output: ""
```

The declarations of the unsafeCall method, name property, and all other methods and properties of type Function are included in the ArkTS *Standard Library*.

3.19 Union Types

Union type is a reference type created as a combination of other types.

The syntax of *union type* is as follows:

```
unionType:
    type '|' type ('|' type)*
    ;
```

The values of a *union* type are valid values of all types the union is created from.

A compile-time error occurs if type in the right-hand side of a union type declaration leads to a circular reference.

A compile-time error occurs if *type* is a function type (see *Function Types*) not enclosed in parentheses.

Typical usage examples of *union* types are represented below:

```
type OperationResult = "Done" | "Not done"
function do_action(): OperationResult {
   if (someCondition) {
      return "Done"
   } else {
      return "Not done"
   }
}

class Cat {
   // ...
}
class Dog {
  // ...
}
class Frog {
   // ...
}
type Animal = Cat | Dog | Frog | number
// Cat, Dog, and Frog are some types (class type or interface type)

let animal: Animal = new Cat()
animal = new Frog()
animal = 42
// One may assign the variable of the union type with any valid value

 enum StringEnum {One = "One", Two = "Two"}

 type Union1 = string | StringEnum // OK, to be reduced during normalization

 type Invalid = string | () => string | number // Compile-time error, function type with␣
↪no parenthesis around
 type Valid1 = string | (() => string) | number // OK
 type Valid21 = string | (() => string | number) // OK
```

Values of particular types can be received from a *union* by using different mechanisms as follows:

```
class Cat { sleep () {}; meow () {} }
class Dog { sleep () {}; bark () {} }
class Frog { sleep () {}; leap () {} }

type Animal = Cat | Dog | Frog

let animal: Animal = new Cat()
if (animal instanceof Frog) {
    // animal is of type Frog here, conversion can be used:
    let frog: Frog = animal as Frog
    frog.leap()
}

animal.sleep () // Any animal can sleep
```

Predefined types are represented by the following example:

```
type Predefined = number | boolean
let p: Predefined = 7
if (p instanceof number) {
   // type of 'p' is number here
}
```

Literal types are represented by the following example:

```
type BMW_ModelCode = "325" | "530" | "735"
let car_code: BMW_ModelCode = "325"
if (car_code == "325"){
   car_code = "530"
} else if (car_code == "530"){
   car_code = "735"
} else {
   // pension :-)
}
```

3.19.1 Union Types Normalization

Union types normalization allows minimizing the number of types within a union type, while keeping type safety. Some types can also be replaced for more general types.

Union type $T_1 | \ldots | T_N$, where $N > 1$, can be formally reduced to type $U_1 | \ldots | U_M$, where $M <= N$, or even to a non-union type *V*. In this latter case *V* can be a predefined value type or a literal type.

The normalization process presumes that the following steps are performed one after another:

1. All nested union types are linearized.
2. All type aliases (if any and except recursive ones) are recursively replaced for non-alias types.
3. Identical types within a union type are replaced for a single type with account to the `readonly` type flag priority.
4. If one type in a union is `string`, then all string literal types (if any) are removed.

5. If one type in a union is `never`, then type `never` is removed.

   This procedure is performed recursively until none of the above steps can can be performed again.

The normalization process results in a normalized union type. The process is represented by the examples below:

```
( T1 | T2) | (T3 | T4) // normalized as T1 | T2 | T3 | T4. Linearization

type A = A[] | string  // No changes. Recursive type alias is kept

type B = number
type C = string
type D = B | C // normalized as number | string. Type aliases are unfolded

number | number // normalized as number. Identical types elimination

(number[]) | (readonly number[]) // normalized as readonly number[]. Readonly version␣
↪wins

"1" | string | number // normalized as  string | number. Literal type value belongs to␣
↪another type values

class Base {}
class Derived extends Base {}
Base | Derived // normalized as Base | Derived (no change)
```

The ArkTS compiler applies normalization while processing union types and handling type inference for array literals (see *Array Type Inference from Types of Elements*).

3.19.2 Access to Common Union Members

Where `u` is a variable of union type $T_1 | \ldots | T_N$, ArkTS supports access to a common member of `u.m` if the following conditions are fulfilled:

- Each T_i is an interface or class type;
- Each T_i has a non-static member with the name `m`; and
- For any T_i, `m` is one of the following:
  - Method or accessor with an equal signature; or
  - Same-type field.

Otherwise, a compile-time error occurs as follows:

```
class A {
    n = 1
    s = "aa"
    foo() {}
    goo(n: number) {}
    static foo () {}
}
class B {
```

(continues on next page)

(continued from previous page)

```
    n = 2
    s = 3.14
    foo() {}
    goo() {}
    static foo () {}
}

let u: A | B = new A

let x = u.n // OK, common field
u.foo() // OK, common method

console.log(u.s) // Compile-time error, field types differ
u.goo() // Compile-time error, signatures differ

type AB = A | B
AB.foo() // Compile-time error, foo() is a static method
```

A compile-time error occurs if in some T_i the name `m` is overloaded (see *Overloading*):

```
class C {
    overload foo { foo1, foo2 }
    foo1(a: number): void {}
    foo2(a: string): void {}
}
class D {
    foo(a: number): void {}
    foo2(a: string): void {}
}

function test(x: C | D) {
    x.foo() // Compile-time error, 'foo' in C is the explicit overload
    x.foo2("aa") // OK, 'foo2' in both C and D is a method
}
```

3.19.3 `Keyof` Types

`Keyof` type is a special form of a union type that is built by using the keyword `keyof`. The keyword `keyof` is applied to a class or an interface type (see *Classes* and *Interfaces*). The resultant new type is a union of names (as string literal types) of all accessible members (see *Accessible*) of the class or the interface type.

The syntax of `keyof` type is presented below:

```
keyofType:
    'keyof' typeReference
    ;
```

A compile-time error occurs if `typeReference` is neither a class nor an interface type. The semantics of type `keyof` is represented by the example below:

```
class A {
   field: number
   method() {}
}
type KeysOfA = keyof A // "field" | "method"
let a_keys: KeysOfA = "field" // OK
a_keys = "any string different from field or method"
  // Compile-time error, invalid value for the type KeysOfA
```

If a class or an interface is empty, then its type `keyof` is equivalent to type `never`:

```
class A {} // Empty class
type KeysOfA = keyof A // never
```

3.20 Nullish Types

ArkTS has *nullish types* that are in fact a specific form of union types (see *Union Types*). The following unions can be used as the type to specify a nullish version of type `T`:

- `T | undefined`;
- `T | null`; or
- `T | undefined | null`.

Note

Using type `T | undefined` in *nullish types* is recommended. The details are discussed in *Type null*.

All predefined types except *Type Any*, and all user-defined types are non-nullish types. Non-nullish types cannot have a `null` or `undefined` value at runtime.

A variable declared to have type `T | null` can hold the values of type `T` and its derived types, or the value `null`.

A variable declared to have type `T | undefined` can hold the values of type `T` and its derived types, or the value `undefined`.

A variable declared to have type `T | null | undefined` can hold values of type `T` and its derived types, and the values `undefined` or `null`.

Nullish type is a reference type (see *Union Types*). A reference that is `null` or `undefined` is called a *nullish value*.

An operation that is safe with no regard to the presence or absence of *nullish values* (e.g., re-assigning one nullish value to another) can be used 'as is' for *nullish types*.

The following nullish-safe options exist for dealing with nullish type `T`:

- Using safe operations:
  - Safe method call (see *Method Call Expression* for details);
  - Safe field access expression (see *Field Access Expression* for details);
  - Safe indexing expression (see *Indexing Expressions* for details);

  - Safe function call (see *Function Call Expression* for details);
- Converting from `T | null` or `T | undefined` to `T`:
  - *Cast Expression*;
  - Ensure-not-nullish expression (see *Ensure-Not-Nullish Expression* for details);
- Supplying a value to be used if a *nullish value* is present:
  - Nullish-coalescing expression (see *Nullish-Coalescing Expression* for details).

Note

Nullish types are not compatible with type `Object`:

```
function nullish (
   o: Object, nullish1: null, nullish2: undefined, nullish3: null|undefined,
   nullish4: AnyClassOrInterfaceType|null|undefined
) {
   o = nullish1 /* compile-time error, type 'null' is not compatible with
                   Object */
   o = nullish2 /* compile-time error, type 'undefined' is not compatible
                   with Object */
   o = nullish3 /* compile-time error, type 'null|undefined' is not
                   compatible with Object */
   o = nullish4 /* compile-time error, type
                   'AnyClassOrInterfaceType|null|undefined' is not
                   compatible with Object */
}
```

3.21 Utility Types

ArkTS supports several built-in types, called *utility* types. Utility types allow constructing new types by adjusting properties of initial types, for which purpose notations identical to generics are used. If the initial types are class or interface, then the resultant utility types are also handled as class or interface types. All utility type names are accessible as simple names (see *Accessible*) in any module across all its scopes. Using these names as user-defined entities causes a compile-time error in accordance with *Declarations*. An alphabetically sorted list of utility types is provided below.

3.21.1 Awaited Utility Type

Type `Awaited<T>` constructs a type which includes no type `Promise`. It is similar to `await` in `async` functions, or to the method `.then()` in *Promises*. Any occurrence of type `Promise` is recursively removed until a generic, a function, an array, or a tuple type is detected. If type `Promise` is not a part of a type `T` declaration, then `Awaited<T>` leaves `T` intact.

If T in Awaited<T> is a type parameter, then subtyping for Awaited<T> is based on the subtyping for T. In other words, Awaited<T> is a subtype of Awaited<U> if T is a subtype of U. The use of type Awaited<T> is represented in the example below:

```
type A = Awaited<Promise<string>>           // type A is string

type B = Awaited<Promise<Promise<number>>>  // type B is number

type C = Awaited<boolean | Promise<number>> // type C is boolean | number

type D = Awaited <Object>                   // type D is Object

type E = Awaited<Promise<Promise<number>|Promise<string>|Promise<boolean>>>
                                            // type E is number|string|boolean

type F = Awaited<Promise<(p: Promise<string>) => Promise<number>>>
                                            // type F is (p: Promise<string>) => Promise
↪<number>>

type G = Awaited<Promise<Array<Promise<number>>>>
                                            // type F is Array<Promise<number>>

function foo <T extends SuperType> (p: Awaited<T>) {}
function bar <T extends SubType> (p: Awaited<T>) {
    foo (p) // is a valid call as Awaited<T extends SubType> <: Awaited<T extends␣
↪SuperType>
}
```

3.21.2 NonNullable Utility Type

Type NonNullable<T> constructs a type by excluding null and undefined types (see *Type null* and *Type void or undefined*). It can be expressed formally as follows (see *Subtyping for Difference Types*):

NonNullable<T> = T - null - undefined.

It implies the following:

- NonNullable<T> leaves T intact if type T contains neither null nor undefined;
- NonNullable<null>, NonNullable<undefined>, or application of NonNullable<> to the union of null and undefined returns never;
- NonNullable<Any> = Any - null - undefined.

The use of type NonNullable<T> is represented in the example below:

```
type X = Object | null | undefined
type Y = NonNullable<X> // Type of 'Y' is Object

class A <T> {
  field: NonNullable<T> // This is a non-nullable version of the type parameter
  constructor (field: NonNullable<T>) {
```

(continues on next page)

(continued from previous page)

```
    this.field = field
  }
}

const a = new A<Object|undefined> (new Object)
a.field // Type of field is Object
```

3.21.3 Partial Utility Type

Type `Partial<T>` constructs a type with all fields (see *Field Declarations*) and properties in their field form (see *Interface Properties*) of `T` set to optional. `T` must be a class or an interface type. Otherwise, a compile-time error occurs. No method (not even any getter or setter) of `T` is a part of the `Partial<T>` type. The use is represented in the example below:

```
interface Issue {
    title: string
    description: string
}

function process(issue: Partial<Issue>) {
    if (issue.title != undefined) {
        /* process title */
    }
}

process({title: "aa"}) // description is undefined
```

In the example above, type `Partial<Issue>` is transformed to a distinct but analogous type as follows:

```
interface /*some name*/ {
    title?: string
    description?: string
}
```

Type `T` is not assignable to `Partial<T>` (see *Assignability*), and no subtyping relation (see *Subtyping*) holds between `T` and `Partial<T>`. Variables of `Partial<T>` are to be initialized with valid object literals.

If `U` is a subtype of `T`, then `Partial<U>` is a subtype of `Partial<T>`.

> **Note**
>
> If class `T` has a user-defined getter, setter, or both, then none of those is called when object literal is used with `Partial<T>` variables. Object literal has its own built-in getters and setters to modify its variables. It is represented in the example below:

```
interface I {
    property: number
```

(continues on next page)

(continued from previous page)

```
}
class A implements I {
    _property: number
    set property(property: number) {
        console.log ("Setter called")
        this._property = property
    }
    get property(): number {
        console.log ("Getter called");
        return this._property
    }
}

function foo (partial: Partial<A>) {
    partial.property = 42 // Setter to be called
    console.log(partial.property) // Getter to be called
}

foo ({property: 1}) // No getter or setter from class A is called
// 42 is printed as object literal has its own setter and getter
```

3.21.4 Required Utility Type

Type `Required<T>` is opposite to `Partial<T>`, and constructs a type with all fields (see *Field Declarations*) and properties in their field form (see *Interface Properties*) of `T` set to required (i.e., not optional). `T` must be a class or an interface type, otherwise a compile-time error occurs. No method (not even any getter or setter) of `T` is part of the `Required<T>` type. Its usage is represented in the example below:

```
interface Issue {
    title?: string
    description?: string
}

let c: Required<Issue> = { // Compile-time error, 'description' should be defined
    title: "aa"
}
```

In the example above, type `Required<Issue>` is transformed to a distinct but analogous type as follows:

```
interface /*some name*/ {
    title: string
    description: string
}
```

Type `T` is not assignable to `Required<T>` (see *Assignability*), and no subtyping relation (see *Subtyping*) holds between `T` and `Required<T>`. Variables of `Required<T>` are to be initialized with valid object literals.

If `U` is a subtype of `T`, then `Required<U>` is a subtype of `Required<T>`.

3.21.5 Readonly Utility Type

Type `Readonly<T>` constructs a type with all fields (see *Field Declarations*) and properties in their field form (see *Interface Properties*) of `T` set to `readonly`. It means that such fields and properties of the constructed type cannot be reassigned. `T` must be a class or an interface type, otherwise a compile-time error occurs. No method (not even any getter or setter) of `T` is part of the `Readonly<T>` type. Its usage is represented in the example below:

```
interface Issue {
    title: string
}

const myIssue: Readonly<Issue> = {
    title: "One"
};

myIssue.title = "Two" // Compile-time error, readonly property
```

Type `T` is assignable (see *Assignability*) to `Readonly<T>`, and allows assignments as a consequence:

```
class A {
   f1: string = ""
   f2: number = 1
   f3: boolean = true
}
let x = new A
let y: Readonly<A> = x // OK
```

Type `T` is a subtype (see *Subtyping*) of `Readonly<T>`.

If type `U` is a subtype of `T`, then `Readonly<U>` is a subtype of `Readonly<T>`.

3.21.6 Record Utility Type

Type `Record<K, V>` constructs a container that maps keys (of type `K`) to values of type `V`.

Type `K` is restricted to numeric types (see *Numeric Types*), type `string`, string literal types, and union types constructed from these types.

A compile-time error occurs if any other type, or literal of any other type is used in place of this type.

Its usage is represented in the example below:

```
type R1 = Record<number, Object>             // OK
type R2 = Record<boolean, Object>            // Compile-time error
type R3 = Record<"salary" | "bonus", Object> // OK
type R4 = Record<"salary" | boolean, Object> // Compile-time error
type R5 = Record<"salary" | number, Object>  // OK
type R6 = Record<string | number, Object>    // OK
```

Type `V` has no restrictions.

A special form of object literals is supported for instances of type `Record` (see *Object Literal of Record Type*).

Access to `Record<K, V>` values is performed by an *indexing expression* like *r[index]*, where *r* is an instance of type `Record`, and *index* is the expression of type `K` (see *Record Indexing Expression* for detail).

Variables of type `Record<K, V>` can be initialized by a valid object literal of Record type (see *Object Literal of Record Type*) where the literal is valid if the type of key expression is compatible with key type `K`, and the type of value expression is compatible with the type `V`.

```
type Keys = 'key1' | 'key2' | 'key3'

let x: Record<Keys, number> = {
    'key1': 1,
    'key2': 2,
    'key3': 4,
}
console.log(x['key2']) // prints 2
x['key2'] = 8
console.log(x['key2']) // prints 8
```

In the example above, `K` is a union of literal types and thus the result of an indexing expression is of type `V`. In this case it is `number`.

3.21.7 ReturnType Utility Type

Type `ReturnType<T>` constructs a new type from the return type of a function type `T` (see *Function Types*). A compile-time error occurs if a non-function type except type `never` is provided. The usage is represented in the example below:

```
type MyString = ReturnType<()=> string> // OK
type Incorrect = ReturnType<string>     // Compile-time error

function foo<P extends Function, R = ReturnType<P>>() {}
/* OK, default type for the second type parameter is the return type of
   the function type provided as the first type argument */

foo<()=>number>()  // R is number

type anAny = ReturnType<Function>  // anAny is Any
type aNever = Return Type<never>   // aNever is never
```

3.21.8 Utility Type Private Fields

Utility types are built on top of other types. Private fields of the initial type stay in the utility type but they are not accessible (see *Accessible*) and cannot be accessed in any way. It is represented in the example below:

```
function foo(): string {  // Potentially some side effect
   return "private field value"
}

class A {
   public_field = 444
   private private_field = foo()
}

function bar (part_a: Readonly<A>) {
   console.log (part_a)
}

bar ({public_field: 777}) // OK, object literal has no field `private_field`
bar ({public_field: 777, private_field: ""}) // Compile-time error, incorrect field name

bar (new A) // OK, object of type Readonly<A> has field `private_field`
```

3.21.9 Nesting Utility Types

If more than one utility types are required then they can be nested as in example below:

```
interface Issue {
  title?: string
}

const myIssue: Required<Readonly<Issue>> = {
   title: "One"
};
console.log(myIssue.title)  // Safe: required property
myIssue.title = "Two" // Compile-time error, readonly property
```

3.22 Default Values for Types

> **Note**
>
> This ArkTS feature is experimental.

So-called *default values* are used by the following types for variables that require no explicit initialization (see *Variable Declarations*):

- *Value Types*;
- Type `undefined` and all its supertypes

All other types, including reference types, enumeration types, and type parameters have no default values.

Default values of value types are as follows:

Data Type	Default Value
number	0 as number
byte	0 as byte
short	0 as short
int	0 as int
long	0 as long
float	+0.0 as float
double	+0.0 as double
char	u0000
boolean	false

Value undefined is the default value of each type to which this value can be assigned.

```
class A {
  f1: string|undefined
  f2?: boolean
}
let a = new A()
console.log (a.f1, a.f2)
// Output: undefined, undefined

let o1: Any
let o2: void
let o3: undefined
console.log (o1, o2, o3)
// Output: undefined, undefined, undefined
```

CHAPTER

FOUR

NAMES, DECLARATIONS AND SCOPES

This chapter introduces the following three mutually-related notions:

- Names,
- Declarations, and
- Scopes.

Each entity in an ArkTS program—a variable, a constant, a class, a type, a function, a method, etc.—is introduced via a *declaration*. An entity declaration defines a *name* of the entity. The name is used to refer to the entity further in the program text. The declaration binds the entity name with the *scope* (see *Scopes*). The scope affects the accessibility of a new entity, and how it can be referred to by its qualified or simple (unqualified) name.

4.1 Names

A name is a sequence of one or more identifiers. A name allows referring to any declared entity. Names can have two syntactical forms:

- *Simple name* that consists of a single identifier;
- *Qualified name* that consists of a sequence of identifiers with the token `'.'` as separator.

Both situations are covered by the below syntax rule:

```
qualifiedName:
  identifier ('.' identifier )*
  ;
```

In a qualified name *N.x* (where *N* is a simple name, and `x` is an identifier that can follow a sequence of identifiers separated with `'.'` tokens), *N* can name the following:

- Name of a module (see *Module Declarations*) that is introduced as a result of `import * as N` (see *Bind All with Qualified Access*) with `x` to name the exported entity;
- A class or interface type (see *Classes*, *Interfaces*) with `x` to name its static member;
- A class or interface type variable with `x` to name its instance member.

4.2 Declarations

A declaration introduces a named entity in an appropriate *declaration scope* (see *Scopes*), see

- *Namespace Declarations*;
- *Type Declarations*;
- *Variable and Constant Declarations*;
- *Function Declarations*;
- *Classes*;
- *Interfaces*;
- *Enumerations*;
- *Constant Or Variable Declarations*;
- *Top-Level Declarations*;
- *Explicit Overload Declarations*;
- *Annotations*;
- *Ambient Declarations*;
- *Accessor Declarations*;
- *Functions with Receiver*.

Each declaration in the declaration scope (see *Scopes*) must be *distinguishable*.

Declarations are *distinguishable* if they have:

- Different names,
- Different signatures (see *Declaration Distinguishable by Signatures*).

Distinguishable declarations are represented by the examples below:

```
const PI = 3.14
const pi = 3
function Pi() {}
type IP = number[]
class A {
    static method() {}
    method() {}
    field: number = PI
    static field: number = PI + pi
}
```

If a declaration is not distinguishable by name, except a valid overload (see *Declaration Distinguishable by Signatures*), then a compile-time error occurs:

```
// Compile-time error, the constant and the function have the same name.
const PI = 3.14
function PI() { return 3.14 }

// Compile-time error, the type and the variable have the same name.
class Person {}
let Person: Person
```

(continues on next page)

(continued from previous page)

```
// Compile-time error, the field and the method have the same name.
class C {
    counter: number
    counter(): number {
      return this.counter
    }
}

/* compile-time error, name of the declaration clashes with the predefined
    type or standard library entity name. */
let number: number = 1
let String = true
function Record () {}
interface Object {}
let Array = 42


/* compile-time error, ambient and non-ambient declarations refer to the
   same entity in a single module
*/
declare function foo()
function foo() {}
```

4.2.1 Declaration Distinguishable by Signatures

The following kinds of same-name declarations are distinguishable by signatures if signatures are not overload-equivalent (see *Overload-Equivalent Signatures*):

- Functions of the same name in the same *declaration scope* (see *Implicit Function Overloading*);
- Methods of the same class with the same name (see *Implicit Method Overloading*);
- Constructors of the same class.

A compile-time error occurs if two same-name functions declared in different scopes are made accessible in one scope:

```
import {foo, goo as bar} from "some module"

function foo() {} // Compile-time error, duplicate declaration
function bar() {} // Compile-time error, duplicate declaration
```

Functions distinguishable by signatures are represented in the example below:

```
  function foo() {}
  function foo(x: number) {}
  function foo(x: number[]) {}
  function foo(x: string) {}
```

Functions with overload-equivalent signatures cause a compile-time error as represented in the following example:

```
// Functions have overload-equivalent signatures
function foo(x: number) {}
function foo(y: number) {}

// Functions have overload-equivalent signatures because of type erasure
function bar(x: Array<number>) {}
function bar(x: Array<string>) {}
```

4.3 Scopes

Different entity declarations introduce new names in different *scopes*. Scope is the region of program text where an entity is declared, along with other regions it can be used in. The following entities are always referred to by their qualified names only:

- Class and interface members (both static and instance ones);
- Entities imported via qualified import; and
- Entities declared in namespaces (see *Namespace Declarations*).

Other entities are referred to by their simple (unqualified) names.

Entities within the scope are accessible (see *Accessible*).

The scope level of an entity depends on the context the entity is declared in:

- *Module level scope* is applicable to modules only. *Constants* and *variables* are accessible (see *Accessible*) from their respective points of declaration to the end of the module. Other entities are accessible through the entire scope level. If exported, a name can be accessed in other modules.
- *Namespace level scope* is applicable to namespaces only. *Constants* and *variables* are accessible (see *Accessible*) from their respective points of declaration to the end of the namespace including all embedded namespaces. Other entities are accessible through the entire namespace scope level including embedded namespaces. If exported, a name can be accessed outside the namespace with mandatory namespace name qualification.
- A name declared inside a class (*class level scope*) is accessible (see *Accessible*) in the class and sometimes, depending on the access modifier (see *Access Modifiers*), outside the class, or by means of a derived class.

  Access to names inside the class is qualified with one of the following:

  - Keywords `this` or `super`;
  - Class instance expression for the names of instance entities; or
  - Name of the class for static entities.

  Outside access is qualified with one of the following:

  - The expression the value stores;
  - A reference to the class instance for the names of instance entities; or
  - Name of the class for static entities.

  ArkTS supports using the same identifier as names of a static entity and of an instance entity. The two names are *distinguishable* by the context, which is either a name of a class for static entities or an expression that denotes an instance.

- A name declared inside an interface (*interface level scope*) is accessible (see *Accessible*) inside and outside that interface (default `public`).
- *The scope of a type parameter* name in a class or interface declaration is that entire declaration, excluding static member declarations.
- The scope of a type parameter name in a function declaration is that entire declaration (*function type parameter scope*).
- The scope of a name declared inside the body of a function or a method declaration is the body of that declaration from the point of declaration and up to the end of the body (*method* or *function scope*). This scope is also applied to function or method parameter names.
- The scope of a name declared inside a block is the body of the block from the point of the name declaration and up to the end of the block (*block scope*).

```
function foo() {
    let x = y // Compile-time error - y is not accessible yet
    let y = 1
}
```

Scopes of two names can overlap (e.g., when statements are nested). If scopes of two names overlap, then:

- The innermost declaration takes precedence; and
- Access to the outer name is not possible.

Class, interface, and enumeration members can only be accessed by applying the dot operator `'.'` to an instance or to a type. Accessing them otherwise is not possible.

4.4 Accessible

Entity is considered accessible if it belongs to the current scope (see *Scopes*) and means that its name can be used for different purposes as follows:

- Type name is used to declare variables, constants, parameters, class fields, or interface properties;
- Function or method name is used to call the function or method;
- Variable name is used to read or change the value of the variable;
- Name of a module introduced as a result of import with Bind All with Qualified Access (see *Bind All with Qualified Access*) is used to deal with exported entities.

4.5 Type Declarations

An interface declaration (see *Interfaces*), a class declaration (see *Classes*), an enum declaration (see *Enumerations*), or a type alias (see *Type Alias Declaration*) are type declarations.

The syntax of *type declaration* is presented below:

```
typeDeclaration:
    classDeclaration
    | interfaceDeclaration
    | enumDeclaration
    | constEnumDeclaration
    | typeAlias
    ;
```

4.5.1 Type Alias Declaration

Type aliases enable using meaningful and concise notations by providing the following:

- Names for anonymous types (array, function, and union types); or
- Alternative names for existing types.

Scopes of type aliases are module or namespace level scopes. Names of all type aliases must follow the uniqueness rules of *Declarations* in the current context.

The syntax of *type alias* is presented below:

```
typeAlias:
    'type' identifier typeParameters? '=' type
    ;
```

Meaningful names can be provided for anonymous types as follows:

```
type Matrix = number[][]
type Handler = (s: string, no: number) => string
type Predicate<T> = (x: T) => boolean
type NullishNumber = number | undefined
```

If the existing type name is too long, then a shorter new name can be introduced by using type alias (particularly for a generic type).

```
type Dictionary = Map<string, string>
type MapOfString<T> = Map<T, string>
```

A type alias acts as a new name only. It neither changes the original type meaning nor introduces a new type.

```
type Vector = number[]
function max(x: Vector): number {
    let m = x[0]
    for (let v of x)
        if (v > m) m = v
    return m
}

let x: Vector = [2, 3, 1]
console.log(max(x)) // output: 3
```

Type aliases can be recursively referenced inside the right-hand side of a type alias declaration.

In a type alias defined as `type A = something`, *A* can be used recursively if it is one of the following:

- Array element type: `type A = A[]`; or
- Type argument of a generic type: `type A = C<A>`.

```
type A = A[] // OK, used as element type

class C<T> { /*body*/}
type B = C<B> // OK, used as a type argument

type D = string | Array<D> // OK
```

Any other use including unresolvable circular references causes a compile-time error, because the compiler does not have enough information about the defined alias:

```
type E = E          // Compile-time error
type F = string | F // Compile-time error
type C<T> = T
type A = C<A>       // Compile-time error
```

The same rules apply to a generic type alias defined as `type A<T> = something`:

```
type A<T> = Array<A<T>> // OK, A<T> is used as a type argument
type A<T> = string | Array<A<T>> // OK

type A<T> = A<T> // Compile-time error
```

A compile-time error occurs if a generic type alias is used without a type argument:

```
type A<T> = Array<A> // Compile-time error
```

Note

There is no restriction on using a type parameter *T* in the right side of a type alias declaration. The following code is valid:

```
type NodeValue<T> = T | Array<T> | Array<NodeValue<T>>;
```

A type parameter of a type alias can *depend* on another type parameter of the same generic.

```
type X<T, S extends T> = T | S
```

A compile-time error occurs if a type parameter in the type parameter section depends on itself directly or indirectly:

```
type X<T extends T> = T // circular dependency
type Y<T extends R, R extends T>  = T // circular dependency
type Z<T extends R, R extends T | undefined> = T // circular dependency
```

4.6 Variable and Constant Declarations

4.6.1 Variable Declarations

A non-ambient *variable declaration* introduces a new variable which is in fact a named storage location. A declared variable must be assigned an initial value before the first usage. The initial value is assigned either as a part of the declaration or in various forms via initialization.

The syntax of *variable declarations* is presented below:

```
variableDeclarations:
    'let' variableDeclarationList
    ;

variableDeclarationList:
    variableDeclaration (',' variableDeclaration)*
    ;

variableDeclaration:
    identifier ':' type initializer?
    | identifier initializer
    ;

initializer:
    '=' expression
    ;
```

When a variable is introduced by a variable declaration, type `T` of the variable is determined as follows:

- `T` is the type specified in a type annotation (if any) of the declaration.
  - If the declaration also has an initializer, then the initializer expression type must be assignable to `T` (see *Assignability with Initializer*).
- If no type annotation is available, then `T` is inferred from the initializer expression (see *Type Inference from Initializer*).

An ambient variable declaration (see *Ambient Constant or Variable Declarations*) must have *type* but no *initializer*. Otherwise, a compile-time error occurs.

```
let a: number // OK
let b = 1 // OK, type 'int' is inferred
let c: number = 6, d = 1, e = "hello" // OK

// OK, type of lambda and type of 'f' can be inferred
let f = (p: number) => b + p
let x // Compile-time error, either type or initializer
```

Every variable in a program must have an initial value before it can be used:

- If the *initializer* of a variable is specified explicitly, then its execution produces the initial value for this variable.
- Otherwise, the following situations are possible:
  - If the type of a variable is `T`, and `T` has a *default value* (see *Default Values for Types*), then the variable is initialized with the default value.
  - If a variable has no default value, then its value must be set by the *Simple Assignment Operator* before any use of the variable.

4.6.2 Constant Declarations

A *constant declaration* introduces a named variable with a mandatory explicit value. The value of a constant cannot be changed by an assignment expression (see *Assignment*). If the constant is an object or array, then object fields or array elements can be modified.

The syntax of *constant declarations* is presented below:

```
constantDeclarations:
    'const' constantDeclarationList
    ;

constantDeclarationList:
    constantDeclaration (',' constantDeclaration)*
    ;

constantDeclaration:
    identifier (':' type)? initializer
    ;
```

The type `T` of a constant declaration is determined as follows:

- If `T` is the type specified in a type annotation (if any) of the declaration, then the initializer expression must be assignable to `T` (see *Assignability with Initializer*).
- If no type annotation is available, then `T` is inferred from the initializer expression (see *Type Inference from Initializer*).

```
const a: number = 1 // OK
const b = 1 // OK, int type is inferred
const c: number = 1, d = 2, e = "hello" // OK
const x // Compile-time error, initializer is mandatory
const y: number // Compile-time error, initializer is mandatory
```

4.6.3 Validity of Initializer

If a variable or constant initializer expression refers to a variable or to a constant, and the declaration of that variable or constant is textually located after the current declaration, then a compile-time error occurs.

```
const a: number = b // Compile-time error
let   b = 1
let   c: number = d // Compile-time error
const d = 123
```

4.6.4 Assignability with Initializer

If a variable or constant declaration contains type annotation `T` and initializer expression *E*, then the type of *E* must be assignable to `T` (see *Assignability*).

4.6.5 Type Inference from Initializer

The type of a declaration that contains no explicit type annotation is inferred from the initializer expression as follows:

- In a variable declaration (not in a constant declaration, though), if the initializer expression is of a literal type, then the literal type is replaced with its supertype, if any (see *Subtyping for Literal Types*). If the initializer expression is of a union type that contains literal types, then each literal type is replaced with its supertype (see *Subtyping for Literal Types*), and then normalized (see *Union Types Normalization*).
- Otherwise, type of a declaration is inferred from its initializer expression.
- Type of an expression with ternary operator `condition ? expr1 : expr2` (see *Ternary Conditional Expressions*) is inferred as follows:
  - If `condition` can be evaluated at compile time, then the type of the entire expression is inferred from `expr1` (where `condition` is `true`) or `expr2` (where `condition` is `false`) in accordance with the rules above.
  - Otherwise, type of the ternary expression is a normalized union of the inferred types of `expr1` and `expr2`.

If type of an initializer expression cannot be inferred, then a compile-time error occurs.

```
// Get boolean value unknown at compile time
function cond(): boolean { return Math.random() < 0.5 ? true : false; }

let a = null                // Type of 'a' is null
let aa = undefined          // Type of 'aa' is undefined
let arr = [null, undefined] // Type of 'arr' is (null | undefined)[]

let b = cond() ? 1 : 2    // Type of 'b' is int

let c = cond() ? 3 : 3.14 // Type of 'c' is int | double
let c1 = true ? 3 : 3.14  // Type of 'c1' is int
let c2 = false ? 3 : 3.14 // Type of 'c1' is double

let d = cond() ? "one" : "two" // Type of 'd' is string

let e = cond() ? 1 : "one" // Type of 'e' is int | string
let e1 = true ? 1 : "one"  // Type of 'e1' is int
let e2 = false ? 1 : "one" // Type of 'e2' is string

const bb  = cond() ? 1 : 2     // Type of 'bb' is int

const cc  = cond() ? 1 : 3.14  // Type of 'cc' is int | double
const cc1 = true   ? 1 : 3.14  // Type of 'cc1' is int
const cc2 = false  ? 1 : 3.14  // Type of 'cc2' is double

const dd  = cond() ? "one" : "two" // Type of 'dd'  is "one" | "two"
```

(continues on next page)

(continued from previous page)

```
const dd1 = true   ? "one" : "two" // Type of 'dd1' is "one"
const dd2 = cond() ? "one" : "two" // Type of 'dd2' is "two"

const ee = cond() ? 1 : "one" // Type of 'ee' is int | "one"

let f = {name: "aa"} // Compile-time error, type of object literal is not inferred

let   x1 = 1 // Type of 'x1' is int
const x2 = 1 // Type of 'x2' is int

let   s1 = "1" // Type of 's1' is string
const s2 = "1" // Type of 's2' is "1"
```

Note

The presence of an initializer for *Ambient Constant or Variable Declarations* causes a compile-time error.

4.7 Function Declarations

Function declarations specify names, signatures, and bodies when introducing *named functions*. An optional function body is a block (see *Block*).

The syntax of *function declarations* is presented below:

```
functionDeclaration:
    modifiers? 'function' identifier
    typeParameters? signature block?
    ;

modifiers:
    'native' | 'async'
    ;
```

Functions must be declared on the top level (see *Top-Level Statements*).

If a function is declared *generic* (see *Generics*), then its type parameters must be specified.

The modifier `native` indicates that the function is a *native function* (see *Native Functions* in Experimental Features). If a *native function* has a body, then a compile-time error occurs.

Functions with the modifier `async` are discussed in *Asynchronous Functions*.

4.7.1 Signatures

A signature defines parameters and the return type (see *Return Type*) of a function, method, or constructor.

The syntax of *signature* is presented below:

```
signature:
    '(' parameterList? ')' returnType?
    ;
```

4.7.2 Parameter List

A signature can contain a *parameter list* that specifies an identifier of each parameter name, and the type of each parameter. The type of each parameter must be defined explicitly. If the *parameter list* is omitted, then the function or the method has no parameters.

The syntax of *parameter list* is presented below:

```
parameterList:
    parameter (',' parameter)* (',' restParameter)? ','?
    | restParameter ','?
    ;

parameter:
    annotationUsage? (requiredParameter | optionalParameter)
    ;

requiredParameter:
    identifier ':' type
    ;
optionalParameter:
    identifier ':' type '=' expression
    | identifier '?' ':' type
    ;
```

If a parameter is *required*, then each function or method call must contain an argument corresponding to that parameter. *Optional Parameters* are described in a separate section. The function below has *required parameters*:

```
function power(base: number, exponent: number): number {
  return Math.pow(base, exponent)
}
power(2, 3) // both arguments are required in the call
```

Several parameters can be *optional*, allowing to omit corresponding arguments in a call (see *Optional Parameters*).

A compile-time error occurs if an *optional parameter* precedes a *required parameter*.

The last parameter of a function or a method can be a single *rest parameter* (see *Rest Parameter*).

If a parameter type is prefixed with `readonly`, then there are additional restrictions on the parameter as described in *Readonly Parameters*.

4.7.3 Readonly Parameters

If the parameter type is `readonly` array or tuple type, then no assignment and no function or method call can modify elements of this array or tuple. Otherwise, a compile-time error occurs:

```
function foo(array: readonly number[], tuple: readonly [number, string]) {
    let element = array[0] // OK, one can get array element
    array[0] = element // Compile-time error, array is readonly

    element = tuple[0] // OK, one can get tuple element
    tuple[0] = element // Compile-time error, tuple is readonly
}
```

Any assignment of readonly parameters and variables must follow the limitations stated in *Type of Expression*.

4.7.4 Optional Parameters

Optional parameters can be of two forms as follows:

```
optionalParameter:
    identifier ':' type '=' expression
    | identifier '?' ':' type
    ;
```

The first form contains an `expression` that specifies a *default value*. It is called *parameter with default value*, and acts as an optional parameter of the type of a function and of its call sites. If a corresponding argument value is `undefined` (i.e., a parameter is omitted, or `undefined` is passed explicitly), then the default value is used:

```
function pair(x: number, y: number = 7)
{
    console.log(x, y)
}
pair(1, 2) // prints: 1 2
pair(1) // prints: 1 7
pair(1, undefined) // prints: 1 7
```

Note

undefined passed as an argument does not affect the *type* of the parameter in a function, a method, or a constructor body.

The second form is a short-cut notation and `identifier '?' ':' type` effectively means that `identifier` has type `T | undefined` with the default value `undefined`.

For example, the following two functions can be used in the same way:

```
function hello1(name: string | undefined = undefined) {}
function hello2(name?: string) {}

hello1() // 'name' has 'undefined' value
hello1("John") // 'name' has a string value
hello2() // 'name' has 'undefined' value
hello2("John") // 'name' has a string value

function foo1 (p?: number) {}
function foo2 (p: number | undefined = undefined) {}

foo1()  // 'p' has 'undefined' value
foo1(5) // 'p' has a numeric value
foo2()  // 'p' has 'undefined' value
foo2(5) // 'p' has a numeric value
```

4.7.5 Rest Parameter

Rest parameters allow functions, methods, constructors, or lambdas to take arbitrary numbers of arguments. *Rest parameters* have the `spread` operator `'...'` as a prefix before the parameter name.

> **Note**
>
> The `spread` operator `'...'` is also used as a prefix in *spread expressions* in ArkTS. The concepts *rest parameter* and *spread expression* are syntactically similar as a result. The difference between the two is clarified in *Spread Expression*.

The syntax of *rest parameter* is presented below:

```
restParameter:
    annotationUsage? '...' identifier ':' type
    ;
```

A compile-time error occurs if a *rest parameter*:

- Is followed by a parameter, which is not a *rest parameter* ;
- Has a type that is not an array type, a tuple type, nor a type parameter constrained by an array or a tuple type.

A call of entity with a *rest parameter* of array type `T[]` (or `FixedArray<T>`) can accept any number of arguments of types that are assignable (see *Assignability*) to `T`:

```
function sum(...numbers: number[]): number {
  let res = 0
  for (let n of numbers)
    res += n
  return res
}

```

(continues on next page)

(continued from previous page)

```
sum() // returns 0
sum(1) // returns 1
sum(1, 2, 3) // returns 6
```

If an argument of array type `T[]` is to be passed to a call of entity with the *rest parameter*, then the spread expression (see *Spread Expression*) must be used with the `spread` operator `'...'` as a prefix before the array argument:

```
function sum(...numbers: number[]): number {
  let res = 0
  for (let n of numbers)
    res += n
  return res
}

let x: number[] = [1, 2, 3]
sum(...x) // spread an array 'x'
   // returns 6
```

A call of entity with a *rest parameter* of tuple type $[T_1, \ldots, T_n]$ can accept only `n` arguments of types that are assignable (see *Assignability*) to the corresponding T_i:

```
function sum(...numbers: [number, number, number]): number {
  return numbers[0] + numbers[1] + numbers[2]
}

sum()          // Compile-time error, wrong number of arguments, 0 instead of 3
sum(1)         // Compile-time error, wrong number of arguments, 1 instead of 3
sum(1, 2, "a") // Compile-time error, wrong type of the 3rd argument
sum(1, 2, 3)   // returns 6
```

It is legal though meaningless to declare a function with an optional parameter followed by a *rest parameter* of a tuple type. However, using such a function without explicitly set optional and *rest parameters* causes compile-time error:

```
// optional tuple + rest tuple
function g(opt?: [number, string], ...tail: [number,string]) { // OK
   // ...
}

g() // Compile-time error, no rest tuple
g([1, "str"]) // Compile-time error, no rest tuple
g([ 1, "str"], 1, "str") // OK
```

If an argument of tuple type $[T_1, \ldots, T_n]$ is to be passed to a call of entity with the rest parameter, then a spread expression (see *Spread Expression*) must have the `spread` operator `'...'` as a prefix before the tuple argument:

```
function sum(...numbers: [number, number, number]): number {
  return numbers[0] + numbers[1] + numbers[2]
}

let x: [number, number, number] = [1, 2, 3]
sum(...x) // spread tuple 'x'
   // returns 6
```

If an argument of fixed-size array type `FixedArray<T>` is to be passed to a function or a method with the rest parameter,

then the spread expression (see *Spread Expression*) must be used with the `spread` operator `'...'` as a prefix before the fixed-size array argument:

```
function sum(...numbers: Array<number>): number {
  let res = 0
  for (let n of numbers)
    res += n
  return res
}

let x: FixedArray<number> = [1, 2, 3]
sum(...x) // spread an fixed-size array 'x'
   // returns 6
```

If constrained by an array or a tuple type, a type parameter can be used with generics as a *rest parameter*.

```
function sum<T extends Array<number>>(...numbers: T): number {
  let res = 0
  for (let n of numbers)
    res += n
  return res
}
```

Note

Any call to a function, method, constructor, or lambda with a rest parameter implies that a new array or tuple is created from the arguments provided.

```
function rest_array(...array_parameter: number[]) {
   // array_parameter is a new array created from the arguments passed
   array_parameter[0] = 1234
   console.log (array_parameter[0]) // 1234 is the output
}
function rest_tuple(...tuple_parameter: [number, string]) {
   // tuple_parameter is a new tuple created from the arguments passed
   tuple_parameter[0] = 1234
   console.log (tuple_parameter[0]) // 1234 is the output
}

const array_argument: number[] =  [1,2,3,4]
const tuple_argument: [number, string] =  [1,"234"]

console.log (array_argument[0], tuple_argument[0]) // 1 1 is the output

rest_array (...array_argument)
     // array_argument is spread into a sequence of its elements

rest_tuple (...tuple_argument)
     // tuple_argument is spread into a sequence of its elements

console.log (array_argument[0], tuple_argument[0]) // 1 1 is the output
```

4.7.6 Shadowing by Parameter

If the name of a parameter is identical to the name of a top-level variable accessible (see *Accessible*) within the body of a function or a method with that parameter, then the name of the parameter shadows the name of the top-level variable within the body of that function or method:

```
let x: number = 1
function foo (x: string) {
    // 'x' refers to the parameter and has type string
}
class SomeClass {
  method (x: boolean) {
    // 'x' refers to the method parameter and has type boolean
  }
}
x++ // 'x' refers to the global variable
```

4.7.7 Return Type

A function, a method, or a lambda return type defines the resultant type of the function, method, or lambda execution (see *Function Call Expression*, *Method Call Expression*, and *Lambda Expressions*). During the execution, the function, method, or lambda can produce a value of a type that is assignable to the return type (see *Assignability*).

The syntax of *return type* is presented below:

```
returnType:
    ':' (type | 'this')
    ;
```

If a function, a method, or a lambda return type is other than `void` or `undefined` (see *Type void or undefined*), or than a union type containing `void` or `undefined`, and the execution path in the function, method, or lambda body has neither a `return` statement (see *return Statements*) nor a `throw` statement (see *throw Statements*), then a compile-time error occurs.

If a function, a method, or a lambda return type is `never` (see *Type never*), and there is an execution path in which all statements execute normally (see *Normal and Abrupt Statement Execution*), then a compile-time error occurs.

A special form of return type with the keyword `this` as type annotation can be used in class instance methods only (see *Methods Returning this*).

If a function, a method, or a lambda return type is not specified, then it is inferred from its body (see *Return Type Inference*). If there is no body, then the function, method, or lambda return type is `void` (see *Type void or undefined*).

```
function foo1 (): number {}  // Compile-time error, return or throw missing
let foo2 =  (): number => {} // Compile-time error, return or throw missing

function foo3 (): undefined {}  // OK, it returns 'undefined' value
let foo4 =  (): undefined => {} // OK, it returns 'undefined' value

```

(continues on next page)

(continued from previous page)

```
function foo5 (): never {}  // Compile-time error, no throw or return never type␣
↪function call
let foo6 =  (): never => {} // Compile-time error, no throw or return never type␣
↪function call

function foo7 (): void {}  // OK, it returns undefined value
let foo8 =  (): void => {} // OK, it returns undefined value

function foo9 () {}   // OK, return type is void and return value is 'undefined'
let foo10 =  () => {} // OK, return type is void and return value is 'undefined'
```

CHAPTER

FIVE

GENERICS

Class, interface, type alias, method, and function are program entities that can be parameterized in ArkTS by one or several types. An entity so parameterized introduces a *generic declaration* (called *a generic* for brevity).

Types used as generic parameters in a generic are called *type parameters* (see *Type Parameters*).

A *generic* must be instantiated in order to be used. *Generic instantiation* is the action that transforms a *generic* into a real program entity (non-generic class, interface, union, array, method, or function), or into another *generic instantiation*. Instantiation (see *Generic Instantiations*) can be performed either explicitly or implicitly.

ArkTS has special types that look similar to generics syntax-wise, and allow creating new types during compilation (see *Utility Types*).

5.1 Type Parameters

Type parameter is declared in the type parameter section. It can be used as an ordinary type inside a *generic*.

Syntax-wise, a *type parameter* is an unqualified identifier with a proper scope (see *Scopes* for the scope of type parameters). Each type parameter can have a *constraint* (see *Type Parameter Constraint*). A type parameter can have a default type (see *Type Parameter Default*), and can specify its *in-* or *out-* variance (see *Type Parameter Variance*).

The syntax of *type parameter* is presented below:

```
typeParameters:
    '<' typeParameterList '>'
    ;

typeParameterList:
    typeParameter (',' typeParameter)*
    ;

typeParameter:
    ('in' | 'out')? identifier constraint? typeParameterDefault?
    ;

constraint:
    'extends' type
    ;
```

(continues on next page)

(continued from previous page)

```
typeParameterDefault:
    '=' typeReference ('[]')?
    ;
```

A generic class, interface, type alias, method, or function defines a set of parameterized classes, interfaces, unions, arrays, methods, or functions respectively (see *Generic Instantiations*). A single type argument can define only one set for each possible parameterization of the type parameter section.

5.1.1 Type Parameter Constraint

If possible instantiations need to be constrained, then an individual *constraint* can be set for each type parameter after the keyword `extends`. A constraint can have the form of any type.

If no constraint is specified, then the constraint is *Type Any*, i.e., the lacking explicit constraint effectively means `extends Any`. As a consequence, the type parameter is not compatible with *Type Object*, and has neither methods nor fields available for use.

If type parameter *T* has type constraint *S*, then the actual type of the generic instantiation must be a subtype of *S* (see *Subtyping*). If the constraint *S* is a non-nullish type (see *Nullish Types*), then *T* is also non-nullish.

```
class Base {}
class Derived extends Base { }
class SomeType { }

class G<T extends Base> { }

let x = new G<Base>      // OK
let y = new G<Derived>   // OK
let z = new G<SomeType>  // Compile-time : SomeType is not compatible with Base

class H<T extends Base|SomeType> {}
let h1 = new H<Base>     // OK
let h2 = new H<Derived>  // OK
let h3 = new H<SomeType> // OK
let h4 = new H<Object>   // Compile-time : Object is not compatible with Base|SomeType

class Exotic<T extends "aa"| "bb"> {}
let e1 = new Exotic<"aa">   // OK
let e2 = new Exotic<"cc">  // Compile-time : "cc" is not compatible with "aa"| "bb"

class A {
  f1: number = 0
  f2: string = ""
  f3: boolean = false
}
class B <T extends keyof A> {}
let b1 = new B<'f1'>    // OK
let b2 = new B<'f0'>    // Compile-time error as 'f0' does not fit the constraint
let b3 = new B<keyof A> // OK
```

A type parameter of a generic can *depend* on an earlier type parameter of the same generic.

```
class G<T, S extends T> {}

class Base {}
class Derived extends Base { }
class SomeType { }

let x: G<Base, Derived>  // OK, the second argument directly
                         // depends on the first one
let y: G<Base, SomeType> // error, SomeType does not depend on Base
```

A compile-time error occurs if a type parameter in the type parameter section depends on itself directly or indirectly:

```
class C<T extends T> {} // circular dependency
class D<T extends R, R extends T> {} // circular dependency
class E<T extends R, R extends T | undefined> {} // circular dependency
```

5.1.2 Type Parameter Default

Type parameters of generic types can have defaults. This situation allows dropping a type argument when a particular type of instantiation is used. However, a compile-time error occurs if:

- A type parameter without a default type follows a type parameter with a default type in the declaration of a generic type;
- Type parameter default refers to a type parameter defined after the current type parameter.

The application of this concept to both classes and functions is presented in the examples below:

```
class SomeType {}
interface Interface <T1 = SomeType> { }
class Base <T2 = SomeType> { }
class Derived1 extends Base implements Interface { }
// Derived1 is semantically equivalent to Derived2
class Derived2 extends Base<SomeType> implements Interface<SomeType> { }

function foo<T = number>(input: T): T { return input}
foo(1) // This call is semantically equivalent to next one
foo<number>(1)

class C1 <T1, T2 = number, T3> {}
// That is a compile-time error, as T2 has default but T3 does not

class C2 <T1, T2 = number, T3 = string> {}
let c1 = new C2<number>          // Equal to C2<number, number, string>
let c2 = new C2<number, string>  // Equal to C2<number, string, string>
let c3 = new C2<number, Object, number> // All 3 type arguments provided

function foo <T1 = T2, T2 = T1> () {}
// Compile-time error,
```

(continues on next page)

(continued from previous page)

```
// as T1's default refers to T2, which is defined after T1
// T2's default is valid as it refers to type parameter T1 already defined
```

5.1.3 Type Parameter Variance

Normally, two different instantiations of the same generic class or interface (like `Array<number>` and `Array<string>`) are handled as different and unrelated types. ArkTS supports type parameter variance that allows *subtyping* relationship between such instantiations (See *Subtyping*), depending on the *subtyping* relationship between argument types.

When declaring *type parameters* of a generic type, special keywords `in` or `out` (called *variance modifiers*) are used to specify the variance of the type parameter (see *Invariance, Covariance and Contravariance*).

Type parameters with the keyword `out` are *covariant*. Covariant type parameters can be used in the out-position only as follows:

- Constructors can have `out` type parameters as parameters.
- Methods can have `out` type parameters as return types.
- Fields that have `out` type parameters as type must be `readonly`.
- Otherwise, a compile-time error occurs.

Type parameters with the keyword `in` are *contravariant*. Contravariant type parameters can be used in the in-position only as follows:

- Methods can have `in` type parameters as parameter types.
- Otherwise, a compile-time error occurs.

Type parameters with no variance modifier are implicitly *invariant*, and can occur in any position.

```
class X<in T1, out T2, T3> {

   // T1 can be used in in-position only
   foo (p: T1) {}  // OK
   foo1(p: T1): T1 { return p } // error, T1 in out-position
   fldT1: T1 // error, T1 in invariant position

  constructor (x: T2) { this.fldT2 = x } // OK
  bar(x: T2) : T2 { return x }           // Compile-time error, x in in-position
  readonly fldT2: T2                     // OK
  bar1() : T2 { return this.fldT2 }      // OK

   // T3 can be used in any position (in-out, write-read)
   fldT3: T3
   method (p: T3): T3 { this.fldT3 = p; return p}  // OK
}
```

In case of function types (see *Function Types*), variance interleaving occurs.

```
class X<in T1, out T2> {
   foo (p: T1): T2 {...}                              // in - out
   foo (p: (p: T2)=> T1) {...}                        // out - in
   foo (p: (p: (p: T1)=>T2)=> T1) {...}               // in - out - in
   foo (p: (p: (p: (p: T2)=> T1)=>T2)=> T1) {...}     // out - in - out - in
   // and further more
}
```

A compile-time error occurs if function or method type parameters have a variance modifier specified.

5.2 Generic Instantiations

As mentioned before, a generic declaration defines a set of corresponding generic or non-generic entities. The process of instantiation is designed to do the following:

- Allow producing new generic or non-generic entities;
- Provide every type parameter with a type argument that can be any kind of type, including the type argument itself.

As a result of the instantiation process, a new class, interface, union, array, method, or function is created.

```
class A <T> {}
class B <U, V> extends A<U> { // A<U> is a new generic type here
    field: A<V>               // A<V> is a new generic type here
    method (p: A<Object>) {}  // A<Object> is a new non-generic type here
}
```

5.2.1 Type Arguments

Type arguments are non-empty lists of types that are used for instantiation.

The syntax of *type arguments* is presented below:

```
typeArguments:
    '<' type (',' type)* '>'
    ;
```

The example below represents instantiations with different forms of type arguments:

```
Array<number>                    // Instantiated with type number
Array<number|string>             // Instantiated with union type
Array<number[]>                  // Instantiated with array type
Array<[number, string, boolean]> // Instantiated with tuple type
Array<()=>void>                  // Instantiated with function type
```

5.2.2 Explicit Generic Instantiations

An explicit generic instantiation is a language construct, which provides a list of *type arguments* (see *Type Arguments*) that specify real types or type parameters to substitute corresponding type parameters of a generic:

```
class G<T> {}    // Generic class declaration
let x: G<number> // Explicit class instantiation, type argument provided

class A {
   method <T> () {}  // Generic method declaration
}
let a = new A()
a.method<string> () // Explicit method instantiation, type argument provided

function foo <T> () {} // Generic function declaration
foo <string> () // Explicit function instantiation, type argument provided

type MyArray<T> = T[] // Generic type declaration
let array: MyArray<boolean> = [true, false] // Explicit array instantiation, type␣
↪argument provided

class X <T1, T2> {}
// Different forms of explicit instantiations of class X producing new generic entities
class Y<T> extends X<number, T> { // class Y extends X instantiated with number and T
   f1: X<Object, T> // X instantiated with Object and T
   f2: X<T, string> // X instantiated with T and string
   constructor() {
     this.f1 = new X<Object,T>
     this.f2 = new X<T,string>
   }
}
```

A compile-time error occurs if type arguments are provided for non-generic class, interface, type alias, method, or function.

In the explicit generic instantiation G <T_1, ..., T_n>, G is the generic declaration, and <T_1, ..., T_n> is the list of its type arguments.

If type parameters T_1, ..., T_n of a generic declaration are constrained by the corresponding C_1, ..., C_n, then T_i is assignable to each constraint type C_i (see *Assignability*). All subtypes of the type listed in the corresponding constraint have each type argument T_i of the parameterized declaration ranging over them.

A generic instantiation G <T_1, ..., T_n> is *well-formed* if **all** of the following is true:

- The generic declaration name is G;
- The number of type arguments equals the number of type parameters of G; and
- All type arguments are assignable to the corresponding type parameter constraint (see *Assignability*).

A compile-time error occurs if an instantiation is not well-formed.

Unless explicitly stated otherwise in appropriate sections, this specification discusses generic versions of class type, interface type, or function.

Any two generic instantiations are considered *provably distinct* if:

- Both are parameterizations of distinct generic declarations; or
- Any of their type arguments is provably distinct.

5.2.3 Implicit Generic Instantiations

In an *implicit* instantiation, type arguments are not specified explicitly. Such type arguments are inferred (see *Type Inference*) from the context in which a generic is referred. If type arguments cannot be inferred, then a compile-time error occurs.

Different cases of type argument inference are represented in the examples below:

```
function foo <G> (x: G, y: G) {} // Generic function declaration
foo (new Object, new Object)     // Implicit generic function instantiation
  // based on argument types: the type argument is inferred

function process <P, R> (arg: P, cb?: (p: P) => R): P | R {
   // Return the data itself or if the processing function provided the
   // result of processing
   return cb != undefined ? cb (arg): arg
}
process (123, () => {}) // P is inferred as 'int', while R is 'void'

function bar <T> (p: number) {}
bar (1) // Compile-time error, type argument cannot be inferred
```

Implicit instantiation is only possible for generic functions and methods.

If a method of a generic class or interface G <T_1, ..., T_n> has its own type parameter `U` with default type (see *Type Parameter Default*) that equals T_i, and an implicit generic instantiation of this method provides no information to infer a type argument, then the type argument correspondent to T_i is used as the type argument for `U`.

This situation is represented in the example below:

```
class A <T> {  // T is the class type parameter
    foo<U = T> (p: U) {} // U is own type parameter with default T
    bar<V = T> () {}     // V is own type parameter with default T
}

// Assume that X1 and X2 are two distinct types
let a = new A<X1>

// Implicit instantiation:
a.foo(new X2) // Type argument is inferred from ``new X2``
a.bar()       // Class type argument X1 is used as no other information is provided

// explicit instantiation:
a.foo<X2> (new X2) // Explicit type argument is used
a.bar<X2> ()       // Explicit type argument is used
```

CHAPTER

SIX

CONTEXTS AND CONVERSIONS

This Chapter defines expression contexts and conversions that can be applied to expressions in different contexts.

Contexts can be of the following kinds:

- *Assignment-like Contexts*;
- *String Operator Contexts* with `string` concatenation (operator `'+'`);
- *Numeric Operator Contexts* with all numeric operators (`'+'`, `'-'`, etc.).

6.1 Assignment-like Contexts

Assignment-like contexts include the following:

- *Declaration contexts* that allow setting an initial value to a variable (see *Variable Declarations*), a constant (see *Constant Declarations*), or a field (see *Field Declarations*) with an explicit type annotation;
- *Assignment contexts* that allow assigning (see *Assignment*) an expression value to a variable;
- *Call contexts* that allow assigning an argument value to a corresponding formal parameter of a function, method, constructor or lambda call (see *Function Call Expression*, *Method Call Expression*, *Explicit Constructor Call*, and *New Expressions*);
- *Return contexts* (see *return Statements*) the allow specifying a resultant value of a function, method or lambda call;
- *Composite literal contexts* that allow setting an expression value to an array element (see *Array Literal Type Inference from Context*), a class, or an interface field (see *Object Literal*);

The examples are presented below:

```
// declaration contexts:
let x: number = 1
const str: string = "done"
class C {
  f: string = "aa"
}

// assignment contexts:
x = str.length
new C().f = "bb"
```

(continues on next page)

(continued from previous page)

```
function foo<T1, T2> (p1: T1, p2: T2) {
  let t1: T1 = p1
  let t2: T2 = p2
}

// call contexts:
function foo(s: string) {}
foo("hello")

// composite literal contexts:
let a: number[] = [str.length, 11]
```

In all these cases, the expression type must be *assignable* to the *target type* (see *Assignability*). *Assignability* allows using of one of *Implicit Conversions*. If there is no applicable conversion, then a compile-time error occurs.

6.2 String Operator Contexts

String context applies only to a non-*string* operand of the binary operator `'+'` if the other operand is `string`.

String conversion for a non-`string` operand is evaluated as follows:

- An operand of an integer type (see *Integer Types and Operations*) is converted to type `string` with a value that represents the operand in the decimal form.
- An operand of a floating-point type (see *Floating-Point Types and Operations*) is converted to type `string` with a value that represents the operand in the decimal form without the loss of information.
- An operand of type `boolean` is converted to type `string` with the values `true` or `false`.
- An operand of enumeration type (see *Enumerations*) is converted to type `string` with the value of the corresponding enumeration member if values of enumeration are of type `string`.
- The operand of a nullish type that has a nullish value is converted as follows:
  - Operand `null` is converted to string `null`.
  - Operand `undefined` is converted to string `undefined`.
- An operand of a reference type or an `enum` type with non-*string* values is converted by applying the method call `toString()`.

If there is no applicable conversion, then a compile-time error occurs.

The target type in this context is always `string`:

```
console.log("" + null) // prints "null"
console.log("value is " + 123) // prints "value is 123"
console.log("BigInt is " + 123n) // prints "BigInt is 123"
console.log(15 + " steps") // prints "15 steps"
let x: string | null | undefined = null
console.log("string is " + x) // prints "string is null"
x = undefined
console.log("string is " + x) // prints "string is undefined"
```

6.3 Numeric Operator Contexts

Numeric contexts apply to the operands of an arithmetic operator. Numeric contexts use numeric types conversions (see *Widening Numeric Conversions*), and ensure that each argument expression can be converted to target type T while the arithmetic operation for the values of type T is being defined.

An operand of enumeration type (see *Enumerations*) can be used in a numeric context if enumeration base type is a numeric type. The type of this operand is assumed to be the same as the enumeration base type.

Numeric contexts take the following forms:

- *Unary Expressions*;
- *Exponentiation Expression*;
- *Multiplicative Expressions*;
- *Additive Expressions*;
- *Shift Expressions*;
- *Relational Expressions*;
- *Equality Expressions*;
- *Bitwise and Logical Expressions*;
- *Conditional-And Expression*;
- *Conditional-Or Expression*.

6.3.1 Numeric Conversions for Relational and Equality Operands

Relational and equality operators (see *Relational Expressions* and *Equality Expressions*) allow *implicit conversion* of numeric type operands of different sizes (see *Widening Numeric Conversions*). Specific details of such conversions are discussed in *Specifics of Numeric Operator Contexts*.

The situation for the relational operator '<' is represented in the example below:

```
let a: int = 1
let b: long = 0

if (b<a) { // 'a' converted to 'long' prior to comparison
    ;
}
```

6.3.2 `char` Conversions for Relational and Equality Operands

Relational and equality operators (see *Relational Expressions* and *Equality Expressions*) allow *implicit conversion* of type `char` operand if other operand is of a numeric type.

A `char` type operand is widened to one of the following:

- Type `int` if the other operand is of types `byte`, `char` or `int`; or
- Type of the other operand.

The situation for the relational operator `'<'` is represented in the example below:

```
function foo(c: char, b: byte, i: int, l: long, d: double) {
    c < b // 'c' and 'b' are converted to 'int' prior to comparison
    c < i // 'c' is converted to 'int' prior to comparison
    c < l // 'c' is converted to 'long' prior to comparison
    c < d // 'c' is converted to 'double' prior to comparison
}
```

6.4 Implicit Conversions

This section describes all implicit conversions that are allowed. Each conversion is allowed in a particular context (e.g., if an expression that initializes a local variable is subject to *Assignment-like Contexts*, then the rules of this context define what specific conversion is implicitly chosen for the expression).

6.4.1 Widening Numeric Conversions

Widening numeric conversion converts a value of a numeric type (see *Numeric Types*) to one of the following:

- Larger numeric type; or
- Union type (see *Widening Numeric to a Union Type*).

This conversion never causes a runtime error.

```
function foo(l: long) {}
function bar(d: double) {}

let b: byte = 1
let s: short = 2
let i: int = 3

foo(b) // byte to long conversion
foo(s) // short to long conversion
foo(i) // int to long conversion

```

(continues on next page)

(continued from previous page)

```
let f: float = 3.14f

bar(i) // int to double conversion
bar(f) // float to double conversion
```

All *widening numeric conversions* are presented in the following table:

From Type	To Type
byte	short, int, long, float, double
short	int, long, float, double
int	long, float, or double
long	float or double
float	double

The above conversions cause no loss of information about the overall magnitude of a numeric value. Some least significant bits of the value can be lost only in conversions from an integer type to a floating-point type if the IEEE 754 *round-to-nearest* mode is used correctly. The resultant floating-point value is properly rounded to the integer value.

6.4.2 Widening Numeric to a Union Type

A numeric value v is converted to U_i of union type (U_1 | ... | U_n), if U_i is a single numeric type in the union that is larger than the value type. Otherwise, a compile-time error occurs.

Note

Before widening to a union type, the following semantic rules are applied in most cases:

- *Type Inference for Constant Expressions* if v is a numeric literal; or
- *Subtyping for Union Types* if U_i is equal to the value type.

If one of these rules applies successfully, then no widening conversion is required.

All cases are represented in the following example:

```
let s: short = 1
let i: int = 2

let u: byte | int = 256 // OK, type inference for numeric literal
console.log(u instanceof int) // output: true

u = i // OK, subtyping
console.log(u instanceof int) // output: true

u = s // OK, widening to union type, short => int conversion
console.log(u instanceof int) // output: true
```

6.4.3 Enumeration to Numeric Type Conversion

If *enumeration base type* is a numeric type, then a value of the enumeration type is converted to one of the following:

- Numeric type equal to or larger than the *enumeration base type*; or
- Union type considering the *enumeration base type* (see *Widening Numeric to a Union Type*).

This conversion never causes a runtime error.

```
enum IntegerEnum {a, b, c}
let int_enum: IntegerEnum = IntegerEnum.b
let int_value: int = int_enum // int_value will get the value 1
let number_value: number = int_enum
   /* number_value will get the value 1 as a result of conversion
      to a larger numeric type */

enum DoubleEnum: double {a = 1.0, b = 2.0, c = 3.141592653589}
let d_enum: DoubleEnum = DoubleEnum.a
let d_value: double = d_enum // d_value will get the value 1.0
```

6.4.4 Enumeration to `string` Type Conversion

A value of *enumeration* with `string` base type is converted to type `string` or to a union type (see *Union Types*) that contains type `string`. This conversion never causes a runtime error.

```
enum StringEnum {a = "a", b = "b", c = "c"}
let s_enum: StringEnum = StringEnum.a
let s: string = s_enum // 's' will get the value of "a"

let u: string | number = s_enum // 'u' will get the value of "a"
```

6.5 Numeric Casting Conversions

A *numeric casting conversion* occurs if the *target type* and the expression type are both `numeric`. The context for a *numeric casting conversion* is where conversion methods are used as defined in the standard library (see *Standard Library*).

The explicit use of methods for *numeric cast conversions* is represented in the following example:

```
function process_int(an_int: int) { /* ... */ }

let pi = 3.14
process_int(pi.toInt())
```

A numeric casting conversion never causes a runtime error.

Numeric casting conversion of an operand of type `double` to target type `float` is performed in compliance with the IEEE 754 rounding rules. This conversion can lose precision or range, resulting in the following:

- Float zero from a nonzero double; and
- Float infinity from a finite double.

Double `NaN` is converted to float `NaN`.

Double infinity is converted to the same-signed floating-point infinity.

A numeric conversion of a floating-point type operand to target types `long` or `int` is performed by the following rules:

- If the operand is `NaN`, then the result is 0 (zero).
- If the operand is positive infinity, or if the operand is too large for the target type, then the result is the largest representable value of the target type.
- If the operand is negative infinity, or if the operand is too small for the target type, then the result is the smallest representable value of the target type.
- Otherwise, the result is the value that rounds toward zero by using IEEE 754 *round-toward-zero* mode.

A numeric casting conversion of a floating-point type operand to types `byte` or `short` is performed in two steps as follows:

- The casting conversion to `int` is performed first (see above);
- Then, the `int` operand is cast to the target type.

A numeric casting conversion from an integer type to a smaller integer type `I` discards all bits except the *N* lowest ones, where *N* is the number of bits used to represent type `I`. This conversion can lose the information on the magnitude of the numeric value. The sign of the resulting value can differ from that of the original value.

CHAPTER

SEVEN

EXPRESSIONS

This Chapter describes the meanings of expressions and the rules for the evaluation of expressions, except the expressions described as experimental (see *Lambda Expressions with Receiver*).

The syntax of *expression* is presented below:

```
expression:
    primaryExpression
    | instanceOfExpression
    | castExpression
    | typeOfExpression
    | nullishCoalescingExpression
    | spreadExpression
    | unaryExpression
    | binaryExpression
    | assignmentExpression
    | ternaryConditionalExpression
    | stringInterpolation
    | lambdaExpression
    | lambdaExpressionWithReceiver
    | awaitExpression
    ;
primaryExpression:
    literal
    | namedReference
    | arrayLiteral
    | objectLiteral
    | recordLiteral
    | thisExpression
    | parenthesizedExpression
    | methodCallExpression
    | fieldAccessExpression
    | indexingExpression
    | functionCallExpression
    | newExpression
    | ensureNotNullishExpression
    ;
binaryExpression:
    multiplicativeExpression
    | exponentiationExpression
    | additiveExpression
    | shiftExpression
```

(continues on next page)

(continued from previous page)

```
    | relationalExpression
    | equalityExpression
    | bitwiseAndLogicalExpression
    | conditionalAndExpression
    | conditionalOrExpression
    ;
```

The syntax below introduces several productions to be used by other expression syntax rules:

```
objectReference:
    typeReference
    | 'super'
    | primaryExpression
    ;
```

`objectReference` refers to one of the following three options:

- Class that is to handle static members;
- `super` that is to access constructors declared in the superclass, or the overridden method version of the superclass;
- *primaryExpression* that is to refer to a variable after evaluation, unless the manner of the evaluation is altered by the chaining operator `'?.'` (see *Chaining Operator*).

If the form of *primaryExpression* is *thisExpression*, then the pattern `this?.` is handled as a compile-time error.

If the form of *primaryExpression* is *super*, then the pattern `super?.` is handled as a compile-time error.

7.1 Operators

An expression is a composition of an operator with its operands. Parentheses are used to change the order of calculation.

Operators, or *operator signs*, are tokens that denote various actions, i.e., addition, subtraction, etc. (see *Operators and Punctuators*) to be performed on the values of *operands*. Depending on the number of operands, operators can be as follows:

- *Unary operator* has a single operand,
- *Binary operator* has two operands, and
- *Ternary operator* has three operands.

Some operators can be both unary and binary.

Each operator has *precedence* and *associativity* which are significant if an expression has several operators:

- *Precedence* defines operator priority, i.e., the order of evaluation of operators with different precedence.
- *Associativity* defines the direction of evaluation (left-to-right or right-to-left) if several operators have the same precedence.

For example, the sum is calculated first due to the higher precedence of `'+'`, and then assignments are processed from right to left due to the right associativity of `'='` in the following chunk of code:

```
let a: number = 1, b: number
b = a = a+10
console.log(b) // prints '11'
```

Note

The parentheses `'( )'` are the *grouping operator* that has the highest precedence and allows changing the order of expression evaluation.

The complete list of operators indicating their precedence and associativity is provided in *Operator Precedence*.

7.1.1 Operator Precedence

The table below summarizes the entire information on the precedence and associativity of operators. Each section on a particular operator also contains detailed information.

Operator	Precedence	Associativity
Grouping	`'()'`	n/a
Member access and chaining	`'.'`, `'?.'`	left-to-right
Access and call	`'[]'`, `'.'`, `'()'`, `new`	n/a
Postfix increment and decrement	`'++'`, `'--'`	n/a
Postfix `'!'` (ensure-not-nullish operator)	`'!'`	n/a
Prefix increment and decrement, unary plus and minus, Prefix `'!'` (logical NOT), bitwise complement, `typeof`, `await`	`'++'`, `'--'`, `'+'`, `'-'`, `'!'`, `'~'`, `typeof`, `await`	n/a
Exponentiation	`'**'`	right-to-left
Multiplicative	`'*'`, `'/'`, `'%'`	left-to-right
Additive	`'+'`, `'-'`	left-to-right
Cast	`as`	left-to-right
Shift	`'<<'`, `'>>'`, `'>>>'`	left-to-right
Relational	`'< >'`, `'<='`, `'>='`, `instanceof`	left-to-right
Equality	`'=='`, `'!='`	left-to-right
Bitwise AND	`'&'`	left-to-right
Bitwise exclusive OR	`'^'`	left-to-right
Bitwise inclusive OR	`'|'`	left-to-right
Logical AND	`'&&'`	left-to-right
Logical OR	`'||'`	left-to-right
Null-coalescing	`'??'`	left-to-right
Ternary	`condition?whenTrue:whenFalse`	right-to-left
Assignment	`'='`, `'+='`, `'-='`, `'%='`, `'*='`, `'/='`, `'&='`, `'^='`, `'|='`, `'<<='`, `'>>='`, `'>>>='`	right-to-left

7.2 Evaluation of Expressions

The result of a program expression *evaluation* denotes the following:

- Variable (the term *variable* is used here in the general, non-terminological sense to denote a modifiable lvalue in the left-hand side of an assignment); or
- Value (results found elsewhere).

A variable or a value are equally considered the *value of the expression* if such a value is required for further evaluation.

The type of an expression is determined at compile time (see *Type of Expression*).

> **Note**
>
> An expression is a sequence of operators and operands that specifies a computation. The evaluation of an expression can produce side effects. Any expression can be a subexpression of a more complex expression. E.g., the function call in *f()*3* is in turn a subexpression of a multiplication expression.

Constant expressions (see *Constant Expressions*) are the expressions with values that can be determined at compile time.

7.2.1 Type of Expression

Every expression in the ArkTS programming language has a type. The type of an expression is determined at compile time.

In most contexts, an expression must be *compatible* with the type expected in a context. This type is called *target type*. If no target type is available in a context, then the expression is called a *standalone expression*:

```
let a = expr // no target type is available

function foo() {
    expr // no target type is available
}
```

Otherwise, the expression is *non-standalone*:

```
let a: number = expr // target type of 'expr' is number

function foo(s: string) {}
foo(expr) // target type of 'expr' is string
```

In some cases, the type of an expression cannot be inferred (see *Type Inference*) from the expression itself (see *Object Literal* as an example). If such an expression is used as a *standalone expression*, then a compile-time error occurs:

```
class P { x: number, y: number }

let x = { x: 10, y: 10 } // standalone object literal - compile time error
let y: P = { x: 10, y: 10 } // OK, type of object literal is inferred
```

The evaluation of an expression type requires completing the following steps:

1. Collect information for type inference (type annotation, generic constraints, etc);
2. Perform *Type Inference*;
3. If the expression type is not yet inferred at a previous step, and the expression is a literal in the general sense, including *Array Literal*, then an attempt is made to evaluate the type from the expression itself.

A compile-time error occurs if none of these steps produces an appropriate expression type.

If the expression type is `readonly`, then the target type must also be `readonly`. Otherwise, a compile-time error occurs:

```
let readonly_array: readonly number[] = [1, 2, 3]

foo1(readonly_array) // OK
foo2(readonly_array) // Compile-time error

function foo1 (p: readonly number[]) {}
function foo2 (p: number[]) {}

let writable_array: number [] = [1, 2, 3]
foo1 (writable_array) // OK, as always safe
```

7.2.2 Normal and Abrupt Completion of Expression Evaluation

The evaluation of an expression requires several steps of computation. An expression is said to be *evaluated normally* if all necessary steps of the evaluation complete with no error thrown. If an error thrown at any step stops the evaluation of an expression, then the situation is defined as an *abrupt completion*. The cause of an *abrupt completion* is explained in the value attached to the error object.

The evaluation of an expression can result in a runtime error as a result of as follows:

- If the value of an array index expression is negative, or greater than, or equal to the length of the array, then an *array indexing expression* (see *Array Indexing Expression*) throws `RangeError`.
- If the type of a value being assigned to a fixed-size array element is not a subtype of an array element type, then an *Assignment* throws *ArrayStoreError*.
- If a *Cast Expression* conversion cannot be performed at runtime, then it throws `ClassCastError`.
- If a right-hand expression has the zero value, then the integer division or integer remainder (see *Division* and *Remainder*) operator throws `ArithmeticError`.

An error during the evaluation of an expression can be caused by a possible hard-to-predict and hard-to-handle linkage and virtual machine error.

Abrupt completion of the evaluation of a subexpression results in the following:

- Immediate abrupt completion of an expression that contains the subexpression (if the evaluation of the contained subexpression is required for the evaluation of the entire expression); and
- Cancellation of all subsequent steps of the normal mode of evaluation.

The terms *complete normally* and *complete abruptly* can also denote normal and abrupt completion of the execution of a statement (see *Normal and Abrupt Statement Execution*). A statement can complete abruptly for many reasons in addition to an error being thrown.

7.2.3 Order of Expression Evaluation

Subexpressions of an expression are evaluated from left to right as represented in the examples below:

- Any right-hand-side expression of a binary operator is evaluated only after the left-hand-side expression is fully evaluated:

```
function arg(s: string): int { console.log(s); return 1; }

arg("left") +  arg("right")
// Prints:
//    left
//    right
```

- The *condition* of a ternary operator is evaluated first. Then, either the subexpression *whenTrue* or the subexpression *whenFalse* is evaluated depending on the *condition* value:

```
function arg(s: string): int { console.log(s); return 1; }

arg("condition True") ? arg("left 1") :  arg("right 1")
arg("condition False")*0 ? arg("left 2") :  arg("right 2")
// Prints:
//    condition True
//    left 1
//    condition False
//    right 2
```

- Operands of *Conditional-And Expression* and *Conditional-Or Expression* are evaluated from left to right. Depending on the value of the left-hand-side operands, the evaluation of the rightmost operand can be skipped:

```
function arg(s: string): int { console.log(s); return 1; }

// Conditional And
arg("left And (true)") && arg("right And")
console.log("---")
arg("left And (false)")*0 && arg("right And")
console.log("---")

// Conditional Or
arg("left Or (true)") && arg("right Or")
console.log("---")
arg("left Or (false)")*0 && arg("right Or")
console.log("---")

// Print
//    left And (true)
//    right And
//    ---
//    left And (false)
//    ---
//    left Or (true)
//    ---
```

(continues on next page)

(continued from previous page)

```
//    left Or (false)
//    right Or
```

- Assignment operator behavior requires additional clarification because the operator is right-associative. An expression containing two assignments is represented in the following example:

```
class A {
   constructor(s:string) { console.log(s); this.tag = s; }
   tag: string
}
function f(s: string) { return new A(s); }

console.log( f("left").tag = f("middle").tag = f("right").tag )
// print
//    left
//    middle
//    right
//    right
```

  The following evaluation steps are performed in the example above:

  – All three subexpressions (left-hand-side, middle, and right-hand-side) are evaluated from left to right. The evaluation of left-hand-side and middle subexpressions results in *lhsExpression*.

  – After that, two assignments are performed from right to left due to right associativity of the assignment operator. The value of the right-hand-side subexpression is assigned to the middle *lhsExpression*, and then the result of the middle expression is assigned to the left *lhsExpression*.

- The ArkTS programming language follows the order of evaluation as indicated explicitly by parentheses, and implicitly by the precedence of operators. This rule particularly applies for infinity and `NaN` values of floating-point calculations. ArkTS considers integer addition and multiplication as provably associative. However, floating-point calculations must not be naively reordered because they are unlikely to be computationally associative (even though they appear mathematically associative).

7.2.4 Evaluation of Arguments

An evaluation of arguments always progresses from left to right up to the first error, or through the end of the expression; i.e., any argument expression is evaluated after the evaluation of each argument expression to its left completes normally (including comma-separated argument expressions that appear within parentheses in method calls, constructor calls, class instance creation expressions, or function call expressions).

If the left-hand argument expression completes abruptly, then no part of the right-hand argument expression is evaluated.

```
function arg(s: string): int { console.log(s); return 1; }
function errArg(s: string, v: int): int { console.log(s); return 1/v }
function test(a: int, b: int, c: int) {}

test(arg("left"), arg("middle"), arg("right"))
test(errArg("errArg (left)", 0), arg("middle"), arg("right"))
// Prints:
```

(continues on next page)

(continued from previous page)

```
//    left
//    middle
//    right
//    errArg (left)
//    ... divideByZero runtime error reported
```

7.2.5 Evaluation of Other Expressions

These general rules cannot cover the order of evaluation of certain expressions when they from time to time cause exceptional conditions. The order of evaluation of the following expressions requires specific explanation:

- Class instance creation expressions (see *New Expressions*);
- *Resizable Array Creation Expressions*;
- *Indexing Expressions*;
- Method call expressions (see *Method Call Expression*);
- Assignments involving indexing (see *Assignment*);
- *Lambda Expressions*.

7.3 Literal

Literals (see *Literals*) denote fixed and unchanging values. Type of a literal is the type of an expression. For numeric literals, type of a literal is inferred using *Type Inference for Constant Expressions*.

7.4 Named Reference

An expression can have the form of a *named reference* as described by the syntax rule as follows:

```
namedReference:
  qualifiedName typeArguments?
  ;
```

Type of a *named reference* expression is the type of the entity to which a *named reference* refers.

QualifiedName (see *Names*) is an expression that consists of dot-separated names. If *qualifiedName* consists of a single identifier, then it is called a *simple name*.

Simple name refers to the following:

- Entity declared in the current module, i.e.,

  - Variable name,
  - Constant name,
  - Function name,
  - Accessor name.
- Variable, constant, or parameter declared in a surrounding block, function, method, or lambda body.

If not a *simple name*, *qualifiedName* refers to the following:

- Entity imported from a module,
- Entity exported from a namespace,
- Member of some class or interface, or
- Special syntax form of *Record Indexing Expression*.

If *typeArguments* are provided, then *qualifiedName* is a valid instantiation of the generic method or function. Otherwise, a compile-time error occurs.

A compile-time error also occurs if a name referred by *qualifiedName* is undefined or inaccessible.

Type of a *named reference* is the type of an expression.

If a *named reference* refers to a function name, it is called *Function Reference*. If a *named reference* refers to a method name, it is called *Method Reference*.

7.4.1 Function Reference

A *function reference* refers to a declared or imported function. Type of a *function reference* is derived from the function signature:

```
function foo(n: number): string { return n.toString() }
let func = foo // type of func is '(n: number) => string'
let x = func(1)  // foo() called via reference
```

A *function reference* can refer to a generic function but only if *Explicit Generic Instantiations* is present, otherwise a compile-time error occurs:

```
function gen<T> (x: T) {}

let a = gen<string> // OK
let b = gen // Compile-time error, no explicit type arguments
```

A compile-time error occurs if the name of an *overloaded function* (see *Implicit Function Overloading*) or the name of an *explicit overload* (see *Explicit Function Overload*) is used as a function reference:

```
function bar(n: number) {}
function bar(s: string) {}

let b = bar // Compile-time error, overloaded function name

function foo1(n: number) {}
function foo2(s: string) {}
```

(continues on next page)

(continued from previous page)

```
overload foo { foo1, foo2 }

foo(1)          // OK, overload call
let x = foo     // Compile-time error, explicit overload name
let y = foo2    // OK, ref to foo2
```

7.4.2 Method Reference

A *method reference* refers to a *static* or *instance* method of a class or an interface. Type of a *method reference* is derived from the method signature:

```
class C {
  static foo(n: number) {}
  bar (s: string): boolean { return true }
}

// Method reference to a static method
const m1 = C.foo  // type of 'm1' is (n: number) => void

// Method reference to an instance method
const m2 = new C().bar // type of 'm1' is (s: string) => boolean
```

If *method reference* refers to an instance method, that the named reference is bounded with the used instance of that class or interface.

```
class C {
    field = 123
    method(): number { return this.field }
}
let c1 = new C
let c2 = new C
let m1 = c1.method // 'c1' is bounded
let m2 = c2.method // 'c2' is bounded
c1.field = 42
console.log (m1(), m2()) // Outputs: 42 123
```

A *method reference* can refer to a generic method only if a generic instantiation is explicitly present (see *Explicit Generic Instantiations*). Otherwise, a compile-time error occurs:

```
class C {
    gen<T> (x: T) {}
}

let a = new C().gen<string> // OK
let b = new C().gen // Compile-time error, no explicit type arguments
```

A compile-time error occurs if the name of an *overloaded method* (see *Implicit Method Overloading*, *Explicit Class Method Overload*, and *Explicit Class Method Overload*) is used as a method reference:

```
class C {
    foo1(n: number) {}
    foo2(s: string) {}
    overload foo { foo1, foo2 }

    function bar(n: number) {}
    function bar(s: string) {}
}

let c = new C()
let f = c.foo // Compile-time error
let b = c.bar // Compile-time error
```

7.5 Array Literal

Array literal is an expression that can be used to create an array or tuple in some cases, with all element values explicitly defined.

The syntax of *array literal* is presented below:

```
arrayLiteral:
    '[' expressionSequence? ']'
    ;

expressionSequence:
    expression (',' expression)* ','?
    ;
```

An *array literal* is a comma-separated list of *initializer expressions* enclosed in square brackets `'['` and `']'`. A trailing comma after the last expression in an array literal is ignored:

```
let x = [1, 2, 3] // OK
let y = [1, 2, 3,] // OK, trailing comma is ignored
```

The number of initializer expressions enclosed in square brackets of the array initializer determines the length of the array to be constructed.

If memory is allocated as required for an array literal, then an array of the specified length is created, and all elements of the array are initialized to the values specified by initializer expressions.

On the contrary, the evaluation of an *array literal* expression completes abruptly if:

- Not enough memory is available for a new array, and `OutOfMemoryError` is thrown; or
- Some initialization expression completes abruptly.

Initializer expressions are executed from left to right. The *n*'th expression specifies the value of the *n-1*'th element of the array.

Array literals can be nested (i.e., the initializer expression that specifies an array element can be an array literal if that element is of array type).

Type of an *array literal expression* is inferred by the following rules:

- If a context is available, then type is inferred from the context. If successful, then type of an array literal is the inferred type. Otherwise, a compile-time error occurs.
- If no context is available, then type is inferred from the types of array literal elements (see *Array Type Inference from Types of Elements*).

More details of both cases are presented below.

7.5.1 Array Literal Type Inference from Context

Type of an array literal can be inferred from the *context* that can be specified as one of the following:

- Explicit type annotation of a variable declaration;
- Left-hand part type of an assignment;
- Explicit return type of a function, a method, or a lambda in a return statement;
- Parameter type in a call;
- Target type of a cast expression; or
- Type of an array element or a class field.

Possible variants are represented in the following example:

```
let a: number[] = [1, 2, 3] // OK, variable type in a declaration is used
a = [4, 5] // OK, variable type is used

let b = [1, 2, 3] as number[]    // OK, cast target type is used

function min(x: number[]): number {
  let m = x[0]
  for (let v of x) {
    if (v < m) m = v
  }
  return m
}
min([1., 3.14, 0.99]); // OK, parameter type is used

// Array of array initialization
type Matrix = number[][]
let m: Matrix = [
    [1, 2], [3, 4] // OK, element type is used
]

class aClass {}
//
let b1: Array <aClass> = [new aClass, new aClass]
let b2: Array <number> = [1, 2, 3]
let b3: FixedArray<number> = [1, 2]
  /* Type of literal is inferred from the context
     taken from b1, b2 and b3 declarations */
```

A compile-time error occurs if the type specified by context is **not** one of the following:

- `Any`;
- `Object`;
- Tuple type;
- Fixed-size array type;
- Value array type;
- Resizable array type;
- Superinterface of a resizable array type;
- Union type that contains at least one constituent type from the above list.

If type used in a context is `Any` or `Object`, then *Array Type Inference from Types of Elements* is used:

```
let a: Object = [1, 2, 3] // OK, array literal is of int[] type
```

If type used in a context is a *tuple type* (see *Tuple Types*), then it is inferred as an array literal type on the following conditions:

- Number of expressions equals the number of constituent types;
- Type of each expression in the array literal is assignable (see *Assignability*) to the constituent type at the respective position.

Otherwise, a compile-time error occurs.

```
let tuple: [number, string] = [1, "hello"] // OK
let incorrect: [number, string] = ["hello", 1] // Compile-time error
```

If type used in a context is a *fixed-size array type* (see *Fixed-Size Array Types*), and type of each expression is assignable to an array element type, then an array literal is of the specified type. Otherwise, a compile-time error occurs.

```
let a: FixedArray<string> = ["hello", "world"] // OK
let b: FixedArray<string> = [1, 2]             // Compile-time error
let c: FixedArray<Object> = [1, "hello"]       // OK
```

If type used in a context is a *value array type* (see *Value Array Types*), and type of each expression is assignable to an array element type, then an array literal is of the specified type. Otherwise, a compile-time error occurs.

```
let a: ValueArray<int> = [1, 2]    // OK
let b: ValueArray<double> = [1, 2] // OK
let b: ValueArray<int> = [3.14]    // Compile-time error
```

If type used in a context is a *resizable array type* (see *Resizable Array Types* and including *Readonly Array Types*), and type of each expression is assignable to an array element type, then an array literal is of the specified type. Otherwise, a compile-time error occurs.

```
let a: Array<string> = ["aa", "bb"]     // OK
let b: string[] = ["aa", "bb"]          // OK
let c: readonly string[] = ["aa", "bb"] // OK

let d: string[] = ["aa", 2]             // Compile-time error

let o: Object[] = ["aa", 2]             // OK
```

If type used in a context is an interface `I`, and:

- If `I` is a generic superinterface of a resizable array type with the single type parameter `I<T>`, then an array literal is considered as an instance of `Array<T>`. If each expression is assignable to `T`, then the array literal is of `I<T>`. Otherwise, a compile-time error occurs;
- If `I` is a non-generic superinterface of a resizable array type, then an array literal type is evaluated by using *Array Type Inference from Types of Elements*, and then inferred as `I`;
- Otherwise, a compile-time error occurs.

This situation is represented in the following example:

```
interface SomeI {}
let a = [1, 2] as SomeI // Compile-time error, SomeI is not a superinterface of Array

let b: ConcatArray<number> = [1, 2]  // OK, instance of Array<number>
let c: ConcatArray<string> = [1, 2]  // Compile-time error, int is not assignable to␣
↪string
let d: ArrayLike<Object> = [1, "aa"] // OK, instance of Array<Object>
```

If a type used in a context is a *union type* (see *Union Types*), then the step *Array Literal Type Inference from Context* is taken repetitively trying to use each type of the *union type* as the context. If only a single type is inferred, then this single type is used as the type of the literal. Otherwise, a compile-time error occurs:

```
let union_1: string[] | [int, int] = [1, 2]
// OK, literal type is a tuple [int, int]

let union_2: number[] | [number, number] = [1, 2]
// Error, as both union types accept literal [1, 2] as valid values

let union_3: (number | string )[] | [(number | string), number] = ["xxx", 2]
// Error, as both union types accept literal ["xxx", 2] as valid values

let union_4: (number | string )[] | [(number | string), number] | string = "xxx"
// OK, as only type string matches the type of "xxx" literal

let union_5: (number | string )[] | [(number | string), number, boolean] | string = [5,␣
↪5]
 // OK, literal type is an array (number | string )[]
```

7.5.2 Array Type Inference from Types of Elements

Where no context is set, and thus the type of an array literal cannot be inferred from the context (see *Type of Expression*), the type of array literal `[ expr`$_1$`, ... , expr`$_N$` ]` is inferred from the initialization expression instead by using the following algorithm:

1. If array literal ($N == 0$) includes no element, then the type of the array literal cannot be inferred, and a compile-time error occurs.
2. If at least one element of an expression type cannot be determined, then the type of the array literal cannot be inferred, and a compile-time error occurs.
3. If all initialization expressions are of the same type `T`, then the array literal type is `T[]`.
4. If each initialization expression is of a numeric type (see *Numeric Types*), then the array literal type is `number[]`.

5. Otherwise, the array literal type is constructed as the array of a union type: `(`T_1 `| ... |` T_N `)[]`, where T_i is the type of $expr_i$, and then:

   - If T_i is a literal type, then it is replaced for its supertype;
   - If T_i is a union type comprised of literal types, then each constituent literal type is replaced for its supertype.
   - *Union Types Normalization* is applied to the resultant union type after the above replacements.

```
type A = number
let u : "A" | "B" = "A"

let a = []                          // Compile-time error, type cannot be inferred
let b = ["a"]                       // type is string[]
let c = [1, 2, 3]                   // type is int[]
let d = [1, 2.1, 3]                 // type is number[]
let e = ["a" + "b", 1, 3.14]        // type is (string | number)[]
let f = [u]                         // type is string[]
let g = [(): void => {}, new A()] // type is (() => void | A)[]
```

7.6 Object Literal

Object literal is an expression that can be used to create a class instance with methods and fields with initial values. In some cases it is more convenient to use an *object literal* in place of a class instance creation expression (see *New Expressions*).

The syntax of *object literal* is presented below:

```
objectLiteral:
   '{' objectLiteralMembers? '}'
   ;

objectLiteralMembers:
   objectLiteralMember (',' objectLiteralMember)* ','?
   ;

objectLiteralMember:
   objectLiteralField | objectLiteralMethod
   ;

objectLiteralField:
   identifier ':' expression
   ;

objectLiteralMethod:
   identifier typeParameters? signature block
   ;
```

An *object literal* is written as a comma-separated list of *object literal members* enclosed in curly braces `'{'` and `'}'`. A trailing comma after the last member is ignored. Each *object literal member* can be either an *object literal field* or an *object literal method* in case when an *object literal* is of interface type.

More details are here *Object Literal of Class Type* and *Object Literal of Interface Type*:

An *object literal field* consists of an identifier and an expression as follows:

```
class Person {
  name: string = ""
  age: number = 0
}
let b: Person = {name: "Bob", age: 25}
let a: Person = {name: "Alice", age: 18, } // OK, trailing comma is ignored
let c: Person | number = {name: "Mary", age: 17} // Literal is of type Person
```

An *object literal method* is a complete declaration of a public method. Examples of object literals with methods are provided in *Object Literal of Interface Type*.

Type of an *object literal expression* is always some class C that is inferred from the context. A type inferred from the context can be either a class (see *Object Literal of Class Type*), or an anonymous class created for the inferred interface type (see *Object Literal of Interface Type*).

A compile-time error occurs if:

- Type of an *object literal* cannot be inferred from the context (see *Type of Expression* for an example);
- Inferred type is not a class or interface type;
- Context is a union type, and an object literal can be treated as a valid value of several union component types;
- New member in an *object literal* is declared;

```
let p = {name: "Bob", age: 25}
        // Compile-time error, type cannot be inferred

class A { field = 1 }
class B { field = 2 }

let q: A | B = {field: 6}
        // Compile-time error, type cannot be inferred as the literal
        // fits both A and B

let u: A = { field: 1, otherField: 2 }
        // Compile-time error, cannot declare a new member in the literal
```

7.6.1 Object Literal of Class Type

If class type C is inferred from the context, then type of an object literal is C:

```
class Person {
  name: string = ""
  age: number = 0
}
function foo(p: Person) { /*some code*/ }
// ...
let p: Person = {name: "Bob", age: 25} /* OK, variable type is
```

(continues on next page)

(continued from previous page)

```
    used */
foo({name: "Alice", age: 18}) // OK, parameter type is used
```

An identifier in each *object literal field* must name a field of class C.

A compile-time error occurs if the identifier does not name an *accessible member field* (see *Accessible*) in type C:

```
class Friend {
  name: string = ""
  protected soname: string = ""
  private nick: string = ""
  age: number
  sex?: "male"|"female"
}
// Compile-time error, nick is private:
let f: Friend = {name: "Alexander", age: 55, nick: "Alex"}
// Compile-time error, soname is protected:
let g: Friend = {name: "Alexander", age: 55, soname: "Reed"}
```

A compile-time error occurs if type of an expression in an *object literal field* is not assignable (see *Assignability*) to the field type:

```
let f: Friend = {name: 123} /* compile-time error, type of right hand-side
is not assignable to the type of the left hand-side */
```

Only class fields that have default values (see *Default Values for Types*) or explicit initializers (see *Variable and Constant Declarations*) can be skipped in an object literal. Otherwise, a compile-time error occurs.

```
let f: Friend = {} /* OK, as name, nick, age, and sex have either default
                      value or explicit initializer */

class C {
    f1: number
    f2: string
    f3!: Object
}
let c1: C = {f2: "xyz", f3: new Object} // OK, f1 type has a default value
let c2: C = {f2: "xyz"} // Compile-time error, f3 value is not provided
```

If type of an object literal is class C, then class C must have an explicit or default *parameterless* constructor, or a constructor with all parameters of the second form of optional parameters (see *Optional Parameters*) that is *accessible* (see *Accessible*) in the class-composite context. Otherwise, a compile-time error occurs.

These situations are presented in the examples below:

```
class C {
  constructor (p: number) {}
}
// ...
let c: C = {} /* compile-time error, no parameterless
       constructor */
```

```
class C {
  private constructor () {}
```

(continues on next page)

(continued from previous page)

```
}
// ...
let c: C = {} /* compile-time error, constructor is not
    accessible */
```

```
class C {
  constructor (p?: number) {}
}
// ...
let c: C = {} // OK as constructor of has an optional parameter
```

```
class C {
}
// ...
let c: C = {} // OK as default constructor is added
```

A compile-time error occurs if an *object literal of class type* explicitly sets readonly fields of a class:

```
class C {
    field1 = 123
    readonly field2: number
    readonly field3: string
    constructor () {
        this.field3 = ""
    }
}

// OK, type ``number`` has default value, field3 set in ctor
let c: C = { field1: 654 }

// Error: field2 and field3 are readonly, cannot be set explicitly
let d: C = { field1: 654, field2: 3, field3: "text" }
```

If a class has accessor (see *Class Accessor Declarations*) in a form of setter then its name can be used as a part of an object literal. Otherwise, a compile-time error occurs:

```
class OK {
    set attr (attr: number) {}
}
const a: OK = {attr: 42} // OK, as the setter be called

class Err {
    get attr (): number { return 42 }
}
const b: Err = {attr: 42} // Compile-time error, no setter for 'attr'
```

7.6.2 Object Literal of Interface Type

If an interface type `I` is inferred from the context, then type of an object literal is an anonymous class implicitly created for interface `I`:

```
interface Person {
  name: string
  set surname(s: string)
  get age(): number
}
let b: Person = {name: "Bob", surname: "Doe", age: 25}
```

In the example above, type of `b` is an anonymous class that contains the same fields as properties of interface `I`. An anonymous class created for the example above has the following fields:

- `name: string`
- `surname: string`
- `age: number`

If a property is defined as a getter, then the type of a field is the return type of the getter. If a property is defined as a setter, then the type of a field is the type of the parameter. If a property is defined as both a getter and a setter, then the parameter type of the setter and the return type of the getter must be the same. Otherwise, a compile-time error occurs as described in *Implementing Required Interface Properties*.

Any properties that are optional can be skipped in an object literal. The values of such optional properties are set to `undefined` as follows:

```
interface Person {
  name: string
  age: number
  sex?: "male"|"female"
}
let b: Person = {name: "Bob", age: 25}
     // 'sex' field has the value 'undefined'
```

Properties that are non-optional cannot be skipped in an object literal, despite some property types having default values (see *Default Values for Types*). If a non-optional property (e.g., *age* in the example above) is skipped, then a compile-time error occurs.

An object literal of interface type must provide an implementation for all interface methods with no default implementation. All such methods are public as in the interface.

```
interface I {
  print_name (name: string): void
  print_something() { console.log ("Something") }
}
let i: I = {
  print_name (name: string) { console.log(name) }
  // No need to implement print_something()
}
i.print_name ("Alice")
i.print_something()
```

Any reference to `this` in an object literal method is a reference to an anonymous class (which is a subtype of the interface) created for the inferred interface type:

Object literal can provide a method with an override-compatible signature (see *Override-Compatible Signatures*):

```
class Base {}
class Drv1 extends Base {}
class Drv2 extends Base {}

interface A {
    foo (p: Drv1): Base
    foo (p: Drv2): Base
}
const a1: A = { foo(p: Base): Drv1 {} }
   /* OK, foo(p: Base) implements both foo (p: Drv1): base and foo (p: Drv2): Base */

const a2: A = { // OK
   foo(p: Drv1): Drv1 { return new Drv1 } // implements foo (p: Drv1): Base
   foo(p: Drv2): Drv2 { return new Drv2 } // implements foo (p: Drv2): Base
}
```

```
interface I { method(i: I): I }
const i: I = { method(i: I): I { return this } }
```

A compile-time error occurs if an object literal of interface type introduces a new method:

```
interface I {}
const i: I = { foo(): void {} } // Compile-time error
```

If an interface has a method implementation, then its object literal can optionally provide a new method implementation. Otherwise, the interface implementation is used:

```
interface I {
   method(): void { console.log ("method() from I is called") }
}

// Valid literal of anonymous class type using interface method
const i1: I = {}
i1.method()

// Valid literal of anonymous class type using own method declaration
const i2: I = {
   method(): void { console.log ("method() from object literal is called") }
}
```

An interface property is set within an object literal by a value. It is independent of the form it is defined in (see *Interface Properties*). The definition within the interface determines how the the property is used:

```
interface I1 {
    set attr (attr: number)
}
const i1: I1 = {attr: 42} // OK, 'attr' is writable property
console.log (i1.attr) // Compile-time error as attr has no getter

interface I2 {
    get attr (): number
}
const i2: I2 = {attr: 42} /* OK, 'attr' is in fact a getter which always returns 42 */
```

(continues on next page)

(continued from previous page)

```
i2.attr = 666 // Compile-time error as attr is readonly
console.log (i2.attr) // OK, output is 42

interface I3 {
    readonly attr: number
}
const i3: I3 = {attr: 42} /* OK, same as above */
i3.attr = 666 // Compile-time error as attr is readonly
console.log (i3.attr) // OK, output is 42

interface I4 {
    attr: number
}
const i4: I4 = {attr: 42} /* OK, getter and setter work with object literal field */
i4.attr = 666 // OK
console.log (i4.attr) // OK
```

7.6.3 Object Literal of Record Type

Generic type `Record<Key, Value>` (see *Record Utility Type*) is used to map properties of a type (type `Key`) to another type (type `Value`). A special form of object literal is used to initialize the value of such type:

The syntax of *record literal* is presented below:

```
recordLiteral:
   '{' keyValueSequence? '}'
   ;

keyValueSequence:
   keyValue (',' keyValue)* ','?
   ;

keyValue:
   expression ':' expression
   ;
```

The first expression in `keyValue` denotes a key and must be of type `Key`. The second expression denotes a value and must be of type `Value`:

```
let map: Record<string, number> = {
    "John": 25,
    "Mary": 21,
}

console.log(map["John"]) // prints 25
```

```
interface PersonInfo {
    age: number
```

(continues on next page)

(continued from previous page)

```
    salary: number
}
let map: Record<string, PersonInfo> = {
    "John": { age: 25, salary: 10},
    "Mary": { age: 21, salary: 20}
}
```

If a key is a union of literal types, then all variants must be listed in the object literal. Otherwise, a compile-time error occurs:

```
let map: Record<"aa" | "bb", number> = {
    "aa": 1,
} // Compile-time error, "bb" key is missing
```

7.6.4 Object Literal Evaluation

The evaluation of an object literal of type `C` (where `C` is either a named class type or an anonymous class type created for the interface) is to be performed by the following steps:

- Call to class `C` constructor with no arguments is executed to initialize an instance `x` of class `C`. The evaluation of the object literal completes abruptly if so does the execution of the constructor.
- All *object literal fields* are then processed from left to right in the textual order they occur in the source code. The following steps are performed for every *object literal field*:
  - Evaluation of the expression; and
  - If successful, then an assignment of the expression value to the corresponding field of `x` as its initial value. Otherwise, the evaluation of the object literal completes abruptly.

The execution of an object literal completes abruptly if so does the execution of at least one of *object literal field* expression.

The evaluation of an object literal completes normally if:

- Class instance was created successfully;
- Class constructor was executed successfully;
- All class instance fields mentioned in the object literal have initial values resulting from the successful execution of all *object literal field* expressions.

7.7 Spread Expression

Spread expression can be used only within an array literal (see *Array Literal*) or argument passing (see *Rest Parameter*). The *expression* must be of an iterable type (see *Iterable Types*) or of a tuple type (see *Tuple Types*).

Otherwise, a compile-time error occurs.

The syntax of *spread expression* is presented below:

```
spreadExpression:
    '...' expression
    ;
```

A *spread expression* is evaluated:

- At compile time by the compiler if *expression* is constant (see *Constant Expressions*);
- Otherwise, at runtime.

Any iterable or tuple object referred by the *expression* is broken down into a sequence of values by the evaluation. This sequence is used where a *spread expression* is used. It can be an assignment, a call of a function, method, or constructor. A sequence of types of these values is the type of the *spread expression*.

Spread expression is one of the two ArkTS concepts that use the `spread` operator `'...'` as a prefix. The difference between *spread expressions* and syntactically similar *rest parameters* is as follows:

- *Spread expression* **generates** a sequence of values. E.g., a sequence can be generated from a type with an iterator.
- *Rest parameter* **receives** a sequence of values and stores the values in an array or a tuple. A *rest parameter* knows nothing about the origin of the values, i.e., the sequence can be a product of a spread operator, or the values can be a result of a direct input.

It is represented in the following example:

```
function f(...a: number[]) {} // Rest, put caller values into an array

f(1,2)    // Thanks to Rest, we can put some values directly ...
let arglist: number[] = [1, 2, 3]
f(...arglist)  // or  say that all values from 'arglist' must be substituted
```

Spread expression of array type supports both *Resizable Array Types* and *Fixed-Size Array Types*. Any combination of *spread expressions* with fixed-size and resizable arrays can be used in an array literal or in a function call as illustrated in the following example:

```
let array1: int[] = [1, 2, 3]
let array2: FixedArray<int> = [4, 5]

// A literal contains two spread expressions with arrays of variable and fixed size
let array3: int[] = [...array1, ...array2] // spread array1 and array2 elements
                                           // while building new array literal at compile time
console.log(array3) // prints [1, 2, 3, 4, 5]

function foo (...array: int[]) {
   console.log (array)
}

// The next two calls are equivalent
foo(...array2)
foo (...[...array2])  // spread array2 elements into arguments of the foo() call

// recall,  'array3 = [ ..array1, ..array2])' (see above)
// next two calls are also equivalent
foo(...array3)
foo(...[...array1, ...array2])

```

(continues on next page)

(continued from previous page)

```
function run_time_spread_application1 (a1: int[], a2: FixedArray<int>) {
   console.log ([...a1, 42, ...a2])
   // Array literal is built at runtime
}
run_time_spread_application1 (array1, array2) // prints [1, 2, 3, 42, 4, 5]
```

Spread expression always copies values from original arrays. A callee changes elements of its own copy, but not elements of arrays used in the call:

```
let a: int[] = [1, 2, 3]

function foo (...p: int[]) {
   p[1] = 4
   console.log ("inside foo()", p)
}
foo(...a)  // prints 'inside foo() 1,4,3'
console.log ("outside foo()", a) // prints 'outside foo() 1,2,3'
```

Since a *spread expression* copies values, the attribute `readonly` of the source array (see *Readonly Array Types*) does not affect an array created by the *spread expression*. If a *spread expression* is to create a readonly target array, then the attribute `readonly` must be used for the target array or for the *rest parameter*:

```
let a: readonly int[] = [1, 2, 3]
let b: int[] = [1, 2, 3]

// 'readonly' array in spread expr, can modify target array Elements
let rw: int[] = [...a]
rw[1] = 1 // OK
function foo(...p_rw: int[]) {
   p_rw[1] = 1 // OK
}

// RW array in spread, readonly target
let ro: readonly int[] = [...b]
   ro[1] = 1 // Compile-time error
function foo(...p_ro: readonly int[]) {
   p_ro[1] = 1 // Compile-time error
}
```

If a *spread expression* is used to pass arguments (see *Rest Parameter*), then the sequence of spread expressions passed as sequential arguments yields a single sequence of values.

```
function accept_spreads_with_rest_parameter (...args: number[]) {
    console.log (args)
}
let arr = [1, 2, 3]
accept_spreads_with_rest_parameter (...arr, ...arr)
   // Output: 1 2 3 1 2 3

function g1() { return [1, 2] }
function g2() { return [3, 4, 5] }
accept_spreads_with_rest_parameter (...g1(), ...g2())
   // Output: 1 2 3 4 5
```

A spread expression for tuples is represented in the example below:

```
let tuple1: [number, string, boolean] = [1, "2", true]
let tuple2: [number, string] = [4, "5"]
 // spread tuple1 and tuple2 elements
let tuple3: [number, string, boolean, number, string] = [...tuple1, ...tuple2]
   // while building new tuple object at compile time
console.log(tuple3) // prints [1, 2, true, 4, 5]

function bar (...tuple: [number, string]) {
  console.log (tuple)
}
bar (...tuple2)  // spread tuple2 elements into arguments of the foo() call

function run_time_spread_application2 (a1: [number, string, boolean], a2: [number,␣
↪string]) {
  console.log ([...a1, 42, ...a2])
    // Such an array literal is built at runtime
}
run_time_spread_application2 (tuple1, tuple2) // prints [1, 2, true, 42, 4, "5"]
```

A spread expression for a class that implements Iterable is represented in the example below:

```
class A<T> implements Iterable<T|undefined> { // variables of type A can be spread
    // To check code with TS, comment line with  `$_iterator()`
    // and uncomment one with `[Symbol.iterator]()`
    $_iterator(): Iterator<T|undefined>  {
    // [Symbol.iterator](): Iterator<T|undefined>  {
      return new MyIteratorResult<T|undefined>(this.data)
    }
    private data: T[]
    constructor (...data: T[]) { this.data = data }
}
class MyIteratorResult <T> implements Iterator<T|undefined> {
    private data: T[]
    private index: number = 0
    next(): IteratorResult<T|undefined> {
        if (this.index >= this.data.length) {
           return MyIteratorResult.end_of_sequence
        } else {
           this.current_element.value = this.data[this.index++]
           return this.current_element
        }
    }
    constructor (data: T[]) { this.data = data }
    private static end_of_sequence: IteratorResult<undefined> = {done: true, value:␣
↪undefined}
    private current_element: IteratorResult<T|undefined> = {done: false, value:␣
↪undefined}
}
function display<T> (...p: T[]) { console.log (p) }
display (... new A<number> (1, 2, 3))        // Spread A with numbers
display (... new A<string> ("aaa", "bbb"))   // Spread A with strings
display (... new A<Object> (1, "aaa", true)) // Spread A with any objects
```

(continues on next page)

(continued from previous page)

```
display (... new A<undefined>)                // Spread A with no objects

type UnionOfIterable = A<number> | new A<string>
function show (...p: UnionOfIterable) { console.log (p) }
show (... new A<number> (1, 2, 3))          // Spread A with numbers
show (... new A<string> ("aaa", "bbb"))     // Spread A with strings
```

Note

If an argument is spread at the call site, then an appropriate parameter must be of the rest kind (see *Rest Parameter*). A compile-time error occurs if an argument is spread into a sequence of ordinary non-optional parameters as follows:

```
function foo1 (n1: number, n2: number) // non-rest parameters
   { ... }
let an_array = [1, 2]
foo1 (...an_array) // Compile-time error

function foo2 (n1: number, n2: string)  // non-rest parameters
   { ... }
let a_tuple: [number, string] = [1, "2"]
foo2 (...a_tuple) // Compile-time error
```

7.8 Parenthesized Expression

The syntax of *parenthesized expression* is presented below:

```
parenthesizedExpression:
    '(' expression ')'
    ;
```

Type and value of a parenthesized expression are the same as those of the contained expression.

7.9 `this` Expression

The syntax of *this expression* is presented below:

```
thisExpression:
    'this'
    ;
```

The keyword `this` can be used as an expression in the body of an instance method of a class (see *Method Body*) or an interface (see *Default Interface Method Declarations*). A corresponding class or interface type is the type of *this*

expression. If a method is declared in an object literal (see *Object Literal*), then the type of the object literal is the type of `this`. A class constructor body can also use `this`, and the exact semantics of such use are described in *Constructor Body*.

The keyword `this` can also be used in the initializer of a class field (see *Field Initialization*).

The keyword `this` can be used in a lambda expression only if it is allowed in the context in which the lambda expression occurs. The type of `this` is the type of a class or an interface in which it is declared, but not the type of the surrounding object literal type, if any.

The keyword `this` in a *direct call* expression `this(` *arguments* `)` can only be used in an explicit constructor call statement (see *Explicit Constructor Call*).

The keyword `this` can also be used in the body of a function with receiver (see *Functions with Receiver*). The type of *this* expression is the declared type of the parameter `this` in a function.

A compile-time error occurs if the keyword `this` appears elsewhere.

The keyword `this` used as a primary expression denotes a value that is a reference to the following:

- Object for which the instance method is called; or
- Object being constructed.

The parameter `this` in a lambda body and in the surrounding context denote the same value.

The class of the actual object referred to at runtime can be `T` if `T` is a class type, or a subclass of `T` (see *Subtyping*) .

The semantics of `this` in different contexts is represented in the example below:

```
interface anInterface {
    method() {
       this // type of 'this' is anInterface
    }
}
class aClass implements anInterface {
    method() {
       this // type of 'this' is aClass
    }
    field = (): void => {
       this // type of 'this' is aClass
    }
}
class AnotherClass {
    anotherMethod() {
        const obj: aClass = { // Object literal
          method () {
              this // type of 'this' is aClass
          },
          field: (): void => {
              this // type of 'this' is AnotherClass
          }
        }
    }
}
```

7.10 Field Access Expression

Field access expression can access a class static field (see *Accessing Static Fields*) or a field of an object to which an object reference refers. An object reference can have different forms as described in detail in *Accessing Current Object Fields* and *Accessing SuperClass Accessors*.

The syntax of *field access expression* is presented below:

```
fieldAccessExpression:
    objectReference ('.' | '?.') identifier
    ;
```

A *field access expression* that contains `'?.'` (see *Chaining Operator*) is called *safe field access* because it handles nullish object references safely.

If object reference evaluation completes abruptly, then so does the entire field access expression.

An object reference used to access a field must be a non-nullish reference type `T`. Otherwise, a compile-time error occurs.

A field access expression is valid if the identifier refers to an accessible member field (see *Accessible*) in type `T`. Otherwise, a compile-time error occurs.

Type of a *field access expression* is the type of a member field.

If the identifier in a *field access expression* denotes the accessor defined for a class or interface type, then either a getter or a setter is called depending on the position of the *field access expression* (see *Class Accessor Declarations* and *Interface Properties* for detail).

7.10.1 Accessing Static Fields

When accessing a static field, *objectReference* takes the form *typeReference*.

The result of a *field access expression* of a static field in a class is one of the following:

- `variable` if the field is not `readonly`. The resultant value can be changed later.
- `value` if the field is `readonly`, except where *field access* occurs in an initializer block (see *Static Initialization*).

7.10.2 Accessing Current Object Fields

When accessing a current object field, *objectReference* takes the form *primaryExpression*.

The process of runtime evaluation of a field access expression starts from the evaluation of *primaryExpression*.

The result of this evaluation can be an instance field or an accessor of a class, or a property of an interface. It is one of the following:

- `variable` if the field is not `readonly`. The resultant value can be changed later;
- `value` if the field is `readonly`, except where *field access* occurs in a constructor (see *Constructor Declaration*);
- call to a proper getter if the *field access expression* is not assigned a new expression; or

- call to a proper setter if the *field access expression* is in the left-hand-side of the assignment.

Only the *primaryExpression* type (not class type of an actual object referred at runtime) is used to determine the field or property to be accessed.

7.10.3 Accessing SuperClass Accessors

The form `super.identifier` is valid to access a superclass accessor (see *Class Accessor Declarations*).

A compile-time error occurs if identifier in 'super.identifier' denotes a field.

```
class Base {
   get property(): number { return 1 }
   set property(p: number) { }
   field = 1234
}
class Derived extends Base {
   get property(): number { return super.property } // OK
   set property(p: number) { super.property = 42 }  // OK
   foo () {
      super.field = 42           // Compile-time error
      console.log (super.field)  // Compile-time error
   }
}
```

7.11 Method Call Expression

A *method call expression* calls

- a static class method;
- an instance method of a class or an interface; or
- a function with receiver (see *Functions with Receiver*).

The syntax of *method call expression* is presented below:

```
methodCallExpression:
    objectReference ('.' | '?.') identifier typeArguments? callArguments
    ;
```

Call arguments are described in *Call Arguments*.

A method call with `'?.'` (see *Chaining Operator*) is called a *safe method call* because it handles nullish values safely.

There are several steps that determine and check the entity to be called at compile time (see *Step 1: Selection of Type to Use*, *Step 2: Selection of Entity to Call*, and *Step 3: Checking Modifiers*).

7.11.1 Step 1: Selection of Type to Use

The *object reference* is used to determine the type in which to search for the method. Three forms of *object reference* are possible:

Form of Object Reference	Type to Use
`typeReference`	Type denoted by `typeReference` must refer to a class. Otherwise, a compile-time error occurs.
`super`	The superclass of the class that contains the method call.
expression of type *T*	`T` if `T` is a class, interface, or union; `T`'s constraint (*Type Parameter Constraint*) if `T` is a type parameter. Otherwise, a compile-time error occurs.

7.11.2 Step 2: Selection of Entity to Call

After the type to use is known (see *Step 1: Selection of Type to Use*), the set of candidates to call is determined by the form of *object reference*, *type to use* and the identifier:

Form of Object Reference	Set of Entities to Call
`typeReference` of type *T*	Static methods named `identifier` of class *T* .
`super`	Instance methods named `identifier` of superclass of the class that contains the call.
expression with *T* type to use	Instance methods named `identifier` of class or interface *T* and *Functions with Receiver* named `identifier` with receiver type *T*. If *T* is a union type, common instance methods see *Access to Common Union Members*.

A compile-time error occurs set is empty, in other words, no entity is available to call.

If a set contains more then one entity, then *Overload Resolution* is used to select the method or function to call (see *Overload Set at Method Call* for details).

Dynamic resolution of method calls is used during program execution to resolve an actual method to be called in case of an instance method in accordance with the method resolved in the step.

7.11.3 Step 3: Checking Modifiers

A single method to call is known at this step. A set of semantic checks for each form of method call must be performed as follows:

- `typeReference.identifier`

  The method must be declared `static`. Otherwise, a compile-time error occurs.

- `expression.identifier`

  The method must not be declared `static`. Otherwise, a compile-time error occurs.

- `super.identifier`

  The method must not be declared `abstract` or `static`. Otherwise, a compile-time error occurs.

Semantic check of a method call is performed in accordance with *Compatibility of Call Arguments*.

7.11.4 Type of Method Call Expression

Type of a *method call expression* is the return type of the method.

```
class A {
   static method() { console.log ("Static method() is called") }
   method()        { console.log ("Instance method() is called") }
}


let x = A.method()     // Compile-time error as void cannot be used as type annotation
A.method ()            // OK
let y = new A().method() // Compile-time error as void cannot be used as type annotation
new A().method()         // OK
```

7.12 Function Call Expression

Function call expression is used to call a function (see *Function Declarations*), a variable of a function type (*Function Types*), or a lambda expression (see *Lambda Expressions*).

The syntax of *function call expression* is presented below:

```
functionCallExpression:
    expression ('?.' | typeArguments)? callArguments
    ;
```

Call arguments are described in *Call Arguments*.

A compile-time error occurs if the expression type is one of the following:

- Different than the function type;
- Nullish type without `'?.'` (see *Chaining Operator*).

If the operator `'?.'` (see *Chaining Operator*) is present, and the *expression* evaluates to a nullish value, then:

- *Call arguments* are not evaluated;
- Call is not performed; and
- Result of *function call expression* is not produced as a consequence.

The function call is *safe* because it handles nullish values properly.

If the form of expression in the call is *qualifiedName*, and *qualifiedName* refers to an *overloaded function* (see *Implicit Function Overloading* and *Explicit Function Overload*), then *Overload Resolution* is used to select the function to call.

A compile-time error occurs if no function is available to call.

Semantic check for a call is performed in accordance with *Compatibility of Call Arguments*.

Various forms of function calls are represented in the example below:

```
function foo() { console.log ("Function foo() is called") }
foo() // function call uses function name to call it

call (foo)            // top-level function passed
call ((): void => { console.log ("Lambda is called") }) // lambda is passed
call (A.method)       // static method
call ((new A).method) // instance method is passed

class A {
   static method() { console.log ("Static method() is called") }
   method() { console.log ("Instance method() is called") }
}

function call (callee: () => void) {
   callee() // function call uses parameter name to call any functional object passed as an argument
}

((): void => { console.log ("Lambda is called") }) () // function call uses lambda expression to call it

let x = foo() // Compile-time error as void cannot be used as type annotation
```

Type of a *function call expression* is the return type of the function.

7.12.1 Call Arguments

The syntax of a *call argument* is presented below:

```
callArguments:
    '(' argumentSequence? ')' trailingLambda?
    ;

argumentSequence:
    expression (',' expression)* ','?
    ;
```

The `callArguments` grammar rule refers to the list of call arguments. Only an argument that corresponds to a *rest parameter* can be a spread expression (see *Spread Expression*).

Trailing lambda call is a special syntactic form of call arguments that contains a *trailing lambda* (see *Trailing Lambdas* for details).

7.13 Indexing Expressions

Indexing expressions are used to access elements of arrays (see *Array Types*), strings (see *Type string*), and `Record` instances (see *Record Utility Type*). Indexing expressions can also be applied to instances of indexable types (see *Indexable Types*).

The syntax of *indexing expression* is presented below:

```
indexingExpression:
    expression ('?.')? '[' expression ']'
    ;
```

Any *indexing expression* has two subexpressions as follows:

- *Object reference expression* before the left bracket; and
- *Index expression* inside the brackets.

If the operator `'?.'` (see *Chaining Operator*) is present in an indexing expression, then:

- If an object reference expression is not of a nullish type, then the chaining operator has no effect.
- Otherwise, object reference expression must be checked against a nullish value. If the value is `undefined` or `null`, then the evaluation of the entire surrounding *primary expression* stops. The result of the entire primary expression is then `undefined`.

If no `'?.'` is present in an indexing expression, then object reference expression must be of array type or `Record` type. Otherwise, a compile-time error occurs.

7.13.1 Array Indexing Expression

Index expression for array indexing must be one of integer types, namely `byte`, `short`, or `int`. Otherwise, a compile-time error occurs.

The conversion of `byte` and `short` types (see *Widening Numeric Conversions*) is performed on an *index expression* to ensure that the resultant type is `int`. Otherwise, a compile-time error occurs.

Other numeric types (`long`, `float`, and `double`/`number`) must be converted explicitly by applying the methods defined in the classes of the *Standard Library*.

```
const a = ["Alice", "Bob", "Carol"]
function demo (l: long, f: float, d: double, n: number) {
    console.log (
       a[l.toInt()], a[f.toInt()],
       a[d.toInt()], a[n.toInt()]
    ) // OK to access array using index expression conversion methods
}
```

If the chaining operator `'?.'` (see *Chaining Operator*) is present, and after its application the type of *object reference expression* is an *array type*, then it makes a valid *array reference expression*, and the type of the array indexing expression is `T`.

The result of an array indexing expression is a variable of type `T` (i.e., an element of the array selected by the value of that *index expression*).

It is essential that, if type T is a reference type, then the fields of array elements can be modified by changing the resultant variable fields:

```
let names: string[] = ["Alice", "Bob", "Carol"]
console.log(names[1]) // prints Bob
names[1] = "Martin"
console.log(names[1]) // prints Martin

console.log (names["1"]) // Compile-time error as index of non-numeric type

class RefType {
    field: number = 42
}
const objects: RefType[] = [new RefType(), new RefType()]
const obj = objects [1]
obj.field = 777            // change the field in the array element
console.log(objects[0].field) // prints 42
console.log(objects[1].field) // prints 777

let an_array = [1, 2, 3]
let element = an_array [3.5] // Compile-time error as index is not integer
function foo (index: number) {
   let element = an_array [index] // Compile-time error as index is not integer
}
```

An array indexing expression evaluated at runtime behaves as follows:

- Object reference expression is evaluated first.
- If the evaluation completes abruptly, then so does the indexing expression, and the index expression is not evaluated.
- If the evaluation completes normally, then the index expression is evaluated. The resultant value of the object reference expression refers to an array.
- If the index expression value of an array is less than zero, greater than or equal to that array's *length*, then `RangeError` is thrown.
- Otherwise, the result of the array access is a type T variable within the array selected by the value of the index expression.

```
function setElement(names: string[], i: int, name: string) {
    names[i] = name // runtime error, if 'i' is out of bounds
}
```

7.13.2 String Indexing Expression

Index expression for string indexing must be of one of integer types, namely `byte`, `short`, or `int`. The same rules apply as in *Array Indexing Expression*.

If the index expression value of a string is less than zero, greater than or equal to that string's *length*, then `RangeError` is thrown.

```
console.log("abc"[1]]) // prints: b
console.log("abc"[3]]) // runtime exception
```

The result of a string indexing expression is a value of `string` type.

> **Note**
>
> String value is immutable, and is not allowed to change a value of a string element by indexing.
>
> ```
> let x = "abc"
> x[1] = "d" // Compile-time error, string value is immutable
> ```

7.13.3 Record Indexing Expression

Indexing expression for a type `Record<Key, Value>` (see *Record Utility Type*) allows getting or setting a value of type `Value` at an index specified by the expression of type `Key`.

The following two cases are to be considered separately:

1. Type `Key` is a union that contains literal types only;
2. Other cases.

Case 1. If type `Key` is a union that contains literal types only, then an *index expression* can only be one of the literals listed in the type. The result of the indexing expression is of type `Value`.

```
type Keys = 'key1' | 'key2' | 'key3'

let x: Record<Keys, number> = {
    'key1': 1,
    'key2': 2,
    'key3': 4,
}
let y = x['key2'] // y value is 2
```

A compile-time error occurs if an index expression is not a valid literal:

```
console.log(x['key4']) // Compile-time error
x['another key'] = 5 // Compile-time error
```

The compiler guarantees that an object of `Record<Key, Value>` for this type `Key` contains values for all `Key` keys.

Case 2. An *index expression* has no restriction. The result of an indexing expression is of type `Value | undefined`.

```
let x: Record<number, string> = {
    1: "hello",
    2: "buy",
}

function foo(n: number): string | undefined {
    return x[n]
```

(continues on next page)

(continued from previous page)

```
}

function bar(n: number): string {
    let s = x[n]
    if (s == undefined) { return "no" }
    return s!
}

foo(3) // prints "undefined"
bar(3) // prints "no"

let y = x[3]
```

Type of *y* in the code above is `string | undefined`. The value of *y* is `undefined`.

An indexing expression evaluated at runtime behaves as follows:

- Object reference expression is evaluated first.
- If the evaluation completes abruptly, then so does the indexing expression, and the index expression is not evaluated.
- If the evaluation completes normally, then the index expression is evaluated. The resultant value of the object reference expression refers to a `record` instance.
- If the `record` instance contains a key defined by the index expression, then the result is the value mapped to the key.
- Otherwise, the result is the literal `undefined`.

Syntactically, the *record indexing expression* can be written by using a field access notation (see *Field Access Expression*) if its *index expression* is formed as an *identifier* of type *string*.

```
type Keys = 'key1' | 'key2' | 'key3'

let x: Record<Keys, number> = {
    'key1': 1,
    'key2': 2,
    'key3': 4,
}
console.log(x.key2) // the same as console.log(x['key2'])
x.key2 = 8          // the same as x['key2'] = 8
console.log(x.key2) // the same as console.log(x['key2'])
```

7.14 Chaining Operator

The *chaining operator* `'?.'` is used to effectively access values of nullish types. It can be used in the following contexts:

- *Field Access Expression*,
- *Method Call Expression*,
- *Function Call Expression*,

- *Indexing Expressions*.

If the value of `expr` in `expr?.` is of a *nullish type*, and is evaluated to `undefined` or `null`, then the evaluation of the entire surrounding *primary expression* stops. The result of the entire primary expression evaluation is then `undefined`. The entire primary expression is then of the union type `undefined | T`, where `T` is a *non-nullish type* of the entire primary expression:

```
class Person {
    name: string
    spouse?: Person = undefined
    constructor(name: string) {
        this.name = name
    }
}

let bob = new Person("Bob")
console.log(bob.spouse?.name) // prints "undefined"
   // type of bob.spouse?.name is undefined|string

bob.spouse = new Person("Alice")
console.log(bob.spouse?.name) // prints "Alice"
   // type of bob.spouse?.name is undefined|string
```

If the value of `expr` in `expr?.` is not of a *nullish type*, then the chaining operator has no effect, and does not influence the type of the entire primary expression:

```
function foo(s1: string, s2: string | undefined) {
    let a = s1?.[0] // 's' is of non-nullish type, type of 'a' is string
    let b = s2?.[0] // type of 'b' is string | undefined
}
```

The chaining operator is allowed in a method call expression for instance methods only. Attempting to use it with a static method is syntactically correct but causes a compile-time error:

```
class A {
   static f(): string {return "" }
   g(): string  { return "" }
}

let s: string|undefined

s = A?.f()            // static method, compile-time error

let b = new A
s = b?.g()            // non-static method, OK
```

A compile-time error occurs if a chaining operator is placed in the context where a variable is expected, e.g., in the left-hand-side expression of an assignment (see *Assignment*) or expression (see *Postfix Increment*, *Postfix Decrement*, *Prefix Increment*, or *Prefix Decrement*).

If an expression preceding a *chaining operator* is known at compile time to always evaluate to a nullish value (`undefined` or `null`) or a non-nullish value at runtime, then a compile-time warning is issued:

```
class C { f = 1}

```

(continues on next page)

(continued from previous page)

```
let c = new C()
c?.f // warning, expression is always non-nullish

let d: C | undefined = undefined
d?.f // warning, expression is always evaluated as undefined
```

7.15 New Expressions

There are two syntactical forms of the *new expression*:

```
newExpression:
    newClassInstance
    | newArrayInstance
    ;
```

Type of a *new expression* is ether `class` or `array`.

A *new class instance expression* creates a new object that is an instance of the specified class and it is described in full details below.

The creation of array instances is an experimental feature discussed in *Resizable Array Creation Expressions*.

The syntax of *new class instance expression* is presented below:

```
newClassInstance:
    'new' typeReference typeArguments? arguments?
    ;
```

Class instance creation expression specifies a class to be instantiated. It optionally lists all actual arguments for the constructor.

```
class A {
   constructor(p: number) {}
}

new A(5) // create an instance and call constructor
const a = new A(6) /* create an instance, call constructor and store
                      created and initialized instance in 'a' */
```

Class instance creation expression can throw an error (see *Error Handling*, *Constructor Declaration*).

A *class instance creation expression* that refers to classes `FixedArray`, `Array`, or classes derived from `Array`, instantiated with an array element type of some class type, is a special form of *array creation expression*.

Attempting to create a *FixedArray* of elements of which the type is a type parameter causes a compile-time error.

```
class A<T> {
   foo (element: T) {
      const a1 = new T[5] (element)  // OK, resizable array with 5 elements
                                     // is created
```

(continues on next page)

(continued from previous page)

```
        const a2 = new FixedArray<T> (5, element) // compile-time error, fixed-size
                                                  // array with 5 elements of type T cannot be
                                                  // created
    }
}
```

This situation is discussed in detail in *Fixed-Size Array Creation* and *Resizable Array Creation Expressions*.

The execution of a class instance creation expression is performed as follows:

- New instance of class is created;
- Constructor of class is called to fully initialize the created instance.

The validity of the constructor call is similar to the validity of the method call as discussed in *Step 2: Selection of Entity to Call*, except the cases discussed in *Constructor Body*.

A compile-time error occurs if `typeReference` is a type parameter.

Note

If a *class instance creation expression* with no argument is used as object reference in a method call expression, then empty parentheses `'( )'` are to be used.

```
class A {  method() {} }

new A.method()   // Compile-time error
new A().method() // OK
(new A).method() // OK
let a = new A    // OK
```

7.16 `instanceof` Expression

The syntax of *instanceof expression* is presented below:

```
instanceOfExpression:
    expression 'instanceof' type
    ;
```

Any `instanceof` expression in the form `expr instanceof T` is of type `boolean`.

The result of an `instanceof` expression is `true` if the *actual type* of evaluated `expr` is a subtype of `T` (see *Subtyping*). Otherwise, the result is `false`.

If type `T` is not *preserved up to undefined* by *Type Erasure*, then a compile-time error occurs.

Generic type (see *Generics*) in the form of *type name* (see *Type References*) can be used as the operand `T` of an `instanceof` expression. In this case, the check is performed against the *erased* type (see *Type Erasure*).

The approach is represented in the following example:

```
class A<T> {}
class B<T> extends A<T> {}

function foo<T>(a: A<T>) {
   let c = a as B<T>   // OK
   let x = new B<string> // OK, explicit type parameter
   console.log(x instanceof B)        // OK
   console.log(x instanceof B<T>)     // Compile-time error, B<T> is not preserved up to␣
↪undefined

   if(a instanceof B) {  // OK, type of instanceof is used for type
                         // checking in `if` clause
      let b = a as B<T>  // OK
   }
}

let a = new B<string>()
foo(a)
```

If an *instanceof expression* is known at compile time to always evaluate to `false` or `true` at runtime, then a compile-time warning is issued:

```
class C {}
class D extends C{}
class E {}

function foo(d: D) {
    console.log(d instanceof C) // warning, expression is always true
    console.log(d instanceof E) // warning, expression is always false
}
```

The `type` of an `instanceof` expression is used for type checking if applicable.

7.17 Cast Expression

The syntax of *cast expression* is as follows:

```
castExpression:
    expression 'as' type
    ;
```

Cast expression in the form `expr as target` applies the *cast operator* `as` to `expr` by issuing the value of a specified `target` type. Thus, the type of a cast expression is always the `target` type.

```
class X {}

let x1 : X = new X()
let ob : Object = x1 as Object // Object is the target type
let x2 : X = ob as X // X is the target type
```

A compile-time error occurs if the `target` type is type `never`:

```
1 as never // Compile-time error
```

If `target` type is not *preserved up to undefined* by *Type Erasure*, then a compile-time warning occurs.

```
class X<out T> { }
function foo(p1: X<Object>) {
    p1 as X<number> // Compile-time warning - such cast converison is type unsafe
}
```

Two specific cases of a *cast expression* are described in the sections below:

- *Type Inference in Cast Expression* if `expr` is a numeric literal (see *Numeric Literals*), an *Array Literal*, or an *Object Literal*;
- *Runtime Checking in Cast Expression* otherwise.

7.17.1 Type Inference in Cast Expression

The following combinations of `expr` and `target` are considered for the `expr as target` expression:

- `expr` is a numeric literal, see *Type Inference for Constant Expressions* for detail;
- `expr` is an *Array Literal*, and `target` is an *array type* or a *tuple type* (see *Array Literal Type Inference from Context* for detail);
- `expr` is an *Object Literal*, and `target` is *class type*, *interface type*, or *Record Utility Type* (see the subsections of *Object Literal* for detail).

This kind of a *cast expression* results in inferring the target type for `expr`. This expression never causes a runtime error by itself. However, the evaluation of array literal elements or object literal properties can cause a runtime error.

Casting for numeric literals is represented in the example below:

```
let x = 1 as byte // OK
let y = 128 as byte // Compile-time error
```

Casting for array literals is represented in the example below:

```
let a = [1, 2] as double[] // OK, [1.0, 2.0]
let b = [1, 2] as double // Compile-time error, wrong target type
let c = [1, "cc"] as double[] // Compile-time error, wrong element type
let d = [1, "cc"] as [double, string] // OK, cast to the tuple type
let e = [1.0, "cc"] as [int, string] // Compile-time error, wrong element type
```

> **Note**
>
> *Assignability* check is applied to the elements of an array literal.

Examples with object literals are provided in *Object Literal*.

7.17.2 Runtime Checking in Cast Expression

If *Type Inference in Cast Expression* cannot be applied, then `expr as target` checks if the type of `expr` is a subtype of `target` (see *Subtyping*).

If the *actual type* of `expr` is a subtype of `target` (see *Subtyping*), then the result of an `as` expression is the result of the evaluated `expr`. Otherwise, `ClassCastError` is thrown.

If `target` type is not *preserved up to undefined* by *Type Erasure*, then a check is performed against the *effective type* of the type. As the *effective type* is less specific than `target` in the case described, the usage of the resulting value can cause type violation, and `ClassCastError` is thrown as a consequence (see *Type Erasure* for detail).

Semantically, a *cast expression* of this kind is coupled tightly with *instanceof Expression* as follows:

- If the result of `x instanceof T` is `true`, then `x as T` succeeds and causes no runtime error;
- If the result of `x instanceof T` is `false`, then `x as T` causes `ClassCastError` thrown at runtime.

This situation is represented in the following example:

```
function foo (x: Object) {
    x as string
}

foo("aa") // OK
foo(1)    // runtime error is thrown in foo by 'as' operator application
```

instanceof Expression can be used to prevent a runtime error. Moreover, the *instanceof Expression* makes *cast conversion* unnecessary in many cases:

```
class Person {
    name: string
    constructor (name: string) { this.name = name }
}

function printName(x: Object) {
    if (x instanceof Person) {
        // no need to cast, access to 'name' is type-safe here
        console.log(x.name)
    } else {
        console.log("not a Person")
    }
}

printName(new Person("Bob")) // output: Bob
printName(1)                 // output: not a Person
```

If the evaluation of a *cast expression* is known at compile time to always succeed or throw `ClassCastError` at runtime, then a compile-time warning is issued:

```
class C {}
class D extends C {}
class E extends C {}

```

(continues on next page)

(continued from previous page)

```
let a: C = new D()
a as D // Compile-time warning, cast always succeeds
a as E // Compile-time warning, cast always throws ClassCastError
```

7.18 typeof Expression

The syntax of *typeof expression* is presented below:

```
typeOfExpression:
    'typeof' expression
    ;
```

Any `typeof` expression is of type `string`.

If *typeof expression* refers to a name of an overloaded function or method, then a compile-time error occurs.

The evaluation of a *typeof expression* starts with the `expression` evaluation. If this evaluation causes an error, then the `typeof` expression evaluation terminates abruptly. Otherwise, the value of a `typeof expression` is defined as follows:

1. The value of a `typeof` expression is known at compile time

Expression Type	typeof Result	Code Example
`string`	"string"	`let s: string = ...` `typeof s`
`boolean`	"boolean"	`let b: boolean = ...` `typeof b`
`bigint`	"bigint"	`let b: bigint = ...` `typeof b`
any class or interface	"object"	`let a: Object = ...` `typeof a`
any function type	"function"	`let f: () => void = ...` `typeof f`
`undefined`, `void`	"undefined"	`typeof undefined` `typeof void`
`null`	"object"	`typeof null`

`T|null`, when `T` is a class (but not Object - see next table), interface or array	"object"	`class C {}`
		`let x: C | null = ...`
		`typeof x`

enumeration type	name of enumeration base type	`enum C {R, G, B}`
		`let c: C = ...`
		`typeof c // "int"`

`number`, `double`	"number"	`let n: number = ...` `typeof n`
Other numeric types: `byte`, `short`, `int`, `long`, `float`	"byte", "short", "int", "long" or "float", depending on type of expression	`let x: byte = ...` `typeof x // "byte"`
`char`	"char"	`let x: char = ...` `typeof x`

2. The value of a `typeof` expression is determined at runtime

The result is the name of an actual type used at runtime for the following expression types:

Expression Type	Code Example

Object	`function f(o: Object) {`
	`  typeof o`
	`}`

union type	`function f(p:A|B) {`
	`  typeof p`
	`}`

type parameter	`class A<T|null|undefined> {`
	`  f: T`
	`  m() {`
	`  typeof this.f`
	`  }`
	`  constructor(p:T) {`
	`  this.f = p`
	`  }`
	`}`

7.19 Ensure-Not-Nullish Expression

Ensure-not-nullish expression is a postfix expression with the operator `'!'`. An *ensure-not-nullish expression* in the expression *e!* checks whether *e* of a nullish type (see *Nullish Types*) evaluates to a nullish value.

The syntax of *ensure-not-nullish expression* is presented below:

```
ensureNotNullishExpression:
    expression '!'
    ;
```

If the result of the evaluation of `expr` in `expr!` is not equal to `null` or `undefined`, then the result of *ensure-not-nullish expression* is the outcome of the evaluation of `expr`, otherwise `NullPointerError` is thrown.

Type of `expr!` is the non-nullish variant of type of `expr`.

If an *ensure-not-nullish expression* is known at compile time to always evaluate at runtime to a non-nullish or a nullish value (`undefined` or `null`), then a compile-time warning is issued. The 'NullPointerError' exception is always thrown at runtime in the latter case as represented below:

```
class C { f = 1}

let c = new C()
c!.f // Compile-time warning, expression is always non-nullish, operator '!' is ignored

let d: C | undefined = undefined
d!.f // Compile-time warning, operator '!' always throws 'NullPointerError', as it is␣
```

(continues on next page)

(continued from previous page)

```
↪applied to nullish value
    // runtime: throws 'NullPointerError'
```

7.20 Nullish-Coalescing Expression

Nullish-coalescing expression is a binary expression that uses the operator `'??'`.

The syntax of *nullish-coalescing expression* is presented below:

```
nullishCoalescingExpression:
    expression '??' expression
    ;
```

A *nullish-coalescing expression* checks whether the evaluation of the left-hand-side expression equals the *nullish* value:

- If so, then the right-hand-side expression evaluation is the result of a nullish-coalescing expression.
- If not so, then the result of the left-hand-side expression evaluation is the result of a nullish-coalescing expression, and the right-hand-side expression is not evaluated (the operator `'??'` is thus *lazy*).

The type of a nullish-coalescing expression is a normalized *union type* (see *Union Types*) formed from the following:

- Non-nullish variant of the type of the left-hand-side expression; and
- Type of the right-hand-side expression.

The semantics of a nullish-coalescing expression is represented in the following example:

```
let x = lhs_expression ?? rhs_expression

let x$ = lhs_expression
if (x$ == null || x == undefined) {x = rhs_expression} else x = x$!

// Type of x is NonNullishType(lhs_expression)|Type(rhs_expression)
```

If the *nullish-coalescing operator* is mixed with a conditional-and or a conditional-or operator without parentheses, then a compile-time error occurs as follows:

```
function  foo(n: boolean | undefined, a: boolean, b: boolean) {
    n ?? a || b   // error, '??' and '||' operations cannot be mixed without parentheses
    n ?? (a || b) // OK
}
```

If the result of a *nullish-coalescing expression* evaluation is known at compile time as a value of the left-hand-side or right-hand-side expression, then a compile-time warning occurs:

```
let a = 1
let b = undefined

a ?? 2 // warning, left-hand-side expression is always used
b ?? 3 // warning, right-hand-side expression is always used
```

7.21 Unary Expressions

The syntax of *unary expression* is presented below:

```
unaryExpression:
    expression '++'
    | expression '--'
    | '++' expression
    | '--' expression
    | '+' expression
    | '-' expression
    | '~' expression
    | '!' expression
    ;
```

All expressions with *unary operators* (except postfix increment and postfix decrement operators) group right-to-left for `'~+x'` to have the same meaning as `'~(+x)'`.

The type of *unaryExpression* is not necessarily the same as the type of the operand provided. The type of *unaryExpression* for each *unary operator* is stated explicitly in the following table:

Unary Operator	Type of Operand	Type of Result
'++', '–' (prefix or postfix)	byte	byte
	short	short
	int	int
	long	long
	float	float
	double	double
	bigint	bigint
'+', '-' (unary)	byte, short, int	int
	long	long
	float	float
	double	double
	bigint	bigint
'~' (bitwise complement)	byte, short, int, float	int
	long, double	long
	bigint	bigint
'!' (logical complement)	boolean or see *Extended Conditional Expressions*	boolean

7.21.1 Postfix Increment

Postfix increment expression is an *expression* followed by the increment operator `'++'`.

The *expression* must be a *left-hand-side expression* (see *Left-Hand-Side Expressions*), and denotes a variable.

A compile-time error occurs if type of the the *expression* is not numeric (see *Numeric Types*) or `bigint`.

Type of a *postfix increment expression* is the type of the variable. The result of a *postfix increment expression* is a value, not a variable.

If the evaluation of the operand *expression* completes normally at runtime, then:

- Value *1* of the same type as a variable is added to the value of the variable; and
- Result of addition is stored back into the variable.

Otherwise, the *postfix increment expression* completes abruptly, and no incrementation occurs.

The value of the *postfix increment expression* is the value of the variable *before* a new value is stored.

The operation of postfix increment is represented in the following code example:

```
let a: short  = 1
let b: float  = 1.5f
let c: bigint = 1n

a++ // result '1', 'a' becomes '2' ('1 + 1')
b++ // result '1.5f', 'b' becomes '2.5f'  ('1.5f + 1f')
c++ // result '1n', 'c' becomes '2n' ('1n + 1n')
```

7.21.2 Postfix Decrement

Postfix decrement expression is an expression followed by the decrement operator `'--'`. The expression must be a *left-hand-side expression* (see *Left-Hand-Side Expressions*). It denotes a variable.

A compile-time error occurs if type of the expression is not numeric (see *Numeric Types*) or `bigint`.

Type of a postfix decrement expression is the type of the variable. The result of a postfix decrement expression is a value, not a variable.

If evaluation of the operand expression completes at runtime, then:

- Value '*1*' of the same type as a variable is subtracted from the value of the variable; and
- Result of the subtraction is stored back into the variable.

Otherwise, the *postfix decrement expression* completes abruptly, and no decrementation occurs.

The value of the *postfix decrement expression* is the value of the variable *before* a new value is stored.

The operation of postfix decrement is represented in the following code example:

```
let a: short  = 1
let b: float  = 1.5f
let c: bigint = 1n

a-- // result '1', 'a' becomes '0' ('1 - 1')
b-- // result '1.5f', 'b' becomes '0.5f'  ('1.5f - 1f')
c-- // result '1n', 'c' becomes '0n' ('1n - 1n')
```

7.21.3 Prefix Increment

Prefix increment expression is an expression preceded by the operator `'++'`. The expression must be a *left-hand-side expression* (see *Left-Hand-Side Expressions*). It denotes a variable.

A compile-time error occurs if the type of the expression is not numeric (see *Numeric Types*) or `bigint`.

Type of a prefix increment expression is the type of the variable. The result of a prefix increment expression is a value, not a variable.

If evaluation of the operand *expression* completes normally at runtime, then:

- Value *1* of the same type as a variable is added to the value of the variable; and
- Result of the addition is stored back into the variable.

Otherwise, the *prefix increment expression* completes abruptly, and no incrementation occurs.

The value of the *prefix increment expression* is the value of the variable *after* a new value is stored.

The operation of prefix increment is represented in the following code example:

```
let a: short  = 1
let b: float  = 1.5f
let c: bigint = 1n

++a // result '2', 'a' becomes '2' ('1 + 1')
++b // result '2.5f', 'b' becomes '2.5f'  ('1.5f + 1f')
++c // result '2n', 'c' becomes '2n' ('1n + 1n')
```

7.21.4 Prefix Decrement

Prefix decrement expression is an expression preceded by the operator `'--'`. The expression must be a *left-hand-side expression* (see *Left-Hand-Side Expressions*). It denotes a variable.

A compile-time error occurs if type of the expression is not numeric (see *Numeric Types*) or `bigint`.

Type of a prefix decrement expression is the type of the variable. The result of a prefix decrement expression is a value, not a variable.

If evaluation of the operand *expression* completes normally at runtime, then:

- Value *1* of the same type as a variable is subtracted from the value of the variable; and
- Result of the subtraction is stored back into the variable.

Otherwise, the *prefix decrement expression* completes abruptly, and no decrementation occurs.

The value of a *prefix decrement expression* is the value of the variable *after* a new value is stored.

The operation of prefix decrement is represented in the following code example:

```
let a: short  = 1
let b: float  = 1.5f
let c: bigint = 1n

--a // result '0', 'a' becomes '0' ('1 - 1')
```

(continues on next page)

(continued from previous page)

```
--b // result '0.5f', 'b' becomes '0.5f'  ('1.5f - 1f')
--c // result '0n', 'c' becomes '0n' ('1n - 1n')
```

7.21.5 Unary Plus

Unary plus expression is an expression preceded by the operator `'+'`. Type of the operand expression with the unary operator `'+'` must be either convertible (see *Implicit Conversions*) to a numeric type (see *Numeric Types*), or of `bigint` type. Otherwise, a compile-time error occurs.

The result of a unary plus expression is always a value, not a variable (even if the result of the operand expression is a variable).

Numeric widening occurs on the *expression* before a *unary plus* operator is applied. The type of the *unary plus* is determined as follows:

- Type of result is `int` for `byte`, `short`, and `int`;
- Type of result is the same as that of the initial *expression* for `long`, `float`, `double`, and `bigint`.

7.21.6 Unary Minus

Unary minus expression is an expression preceded by the operator `'-'`. Type of an operand expression with the unary operator `'-'` must be either convertible (see *Widening Numeric Conversions*) to a numeric type (see *Numeric Types*), or of `bigint` type. Otherwise, a compile-time error occurs.

Numeric widening occurs on the *expression* before a *unary minus* operator is applied. The type of the *unary minus* is determined as follows:

- Type of result is *int* for `byte`, `short`, and `int`;
- Type of result is the same as that of the initial *expression* for `long`, `float`, `double`, and `bigint`.

The result of a unary minus expression is a value, not a variable (even if the result of the operand expression is a variable).

The unary negation operation is always performed on, and the result is drawn from the same value set as the promoted operand value.

Further value set conversions are then performed on the same result.

The value of a unary minus expression at runtime is the arithmetic negation of the promoted value of the operand.

The negation of integer values is the same as subtraction from zero. The ArkTS programming language uses two's-complement representation for integers. The range of two's-complement value is not symmetric. The same maximum negative number results from the negation of the maximum negative *int* or *long*. In that case, an overflow occurs but throws no error. For any integer value *x*, *-x* is equal to *(~x)+1*.

The negation of bigint values and subtraction from the value *0n* are the same.

The negation of floating-point values is *not* the same as subtraction from zero (if *x* is *+0.0*, then *0.0-x* is *+0.0*, however *-x* is *-0.0*).

A unary minus merely inverts the sign of a floating-point number. Special cases to consider are as follows:

- Operand `NaN` results in `NaN` (`NaN` has no sign).
- Operand infinity results in the infinity of the opposite sign.
- Operand zero results in zero of the opposite sign.

7.21.7 Bitwise Complement

Bitwise complement operator `'~'` is applied to an operand of a numeric type or type `bigint`.

If the type of the operand is `double` or `float`, then it is truncated first to `long` or `int`, respectively. If the type of the operand is `byte` or `short`, then the operand is widened to `int`. If the type of the operand is `bigint`, then no conversion is required. Type of result is determined as follows:

- `int` for `byte`, `short`, `int`, and `float`.
- `long` for `long` and `double`.
- `bigint` for `bigint`.

The result of a unary bitwise complement expression is a value, not a variable (even if the result of the operand expression is a variable).

The value of a unary bitwise complement expression at runtime is the bitwise complement of the value of the operand. In all cases, *~x* equals *(-x)-1*.

It is represented by the following example:

```
let b: byte  = 2
let s: short  = 2
let i: int = 2
let f: float = 2.0f

let l: long  = 2
let d: double  = 2.0

let B: bigint = 2n

let rb = ~b
console.log(rb, typeof rb) // prints '-3 int'
let rs = ~s
console.log(rs, typeof rs) // prints '-3 int'
let ri = ~i
console.log(ri, typeof ri) // prints '-3 int'
let rf = ~f
console.log(rf, typeof rf) // prints '-3 int'

let rl = ~l
console.log(rl, typeof rl) // prints '-3 long'
let rd = ~d
console.log(rd, typeof rd) // prints '-3 long'
```

(continues on next page)

(continued from previous page)

```
let rB = ~B
console.log(rB, typeof rB) // prints '-3 bigint'
```

7.21.8 Logical Complement

Logical complement expression is an expression preceded by the operator `'!'`. Type of the operand expression with the unary `'!'` operator must be `boolean` or type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

The unary logical complement expression type is `boolean`.

The value of a unary logical complement expression is `true` if the (possibly converted) operand value is `false`, and `false` if the operand value (possibly converted) is `true`.

7.22 Binary Expressions

The syntax of *binary expression* is presented below:

```
binaryExpression:
    multiplicativeExpression
    | exponentiationExpression
    | additiveExpression
    | shiftExpression
    | relationalExpression
    | equalityExpression
    | bitwiseAndLogicalExpression
    | conditionalAndExpression
    | conditionalOrExpression
    ;
```

Every *binaryExpression* has the form *expression 1* `op` *expression 2*, where `op` is a binary operator (operator sign), and *expression 1* and *expression 2* are its operands.

The subgroups of binary expressions are described further in this chapter.

Possible combinations of types of *expression 1* and *expression 2*, as well as the type of the resulting *binaryExpression* are presented in the table below. Type combinations not listed in the table cause a compile-time error if a combination is detected at compile time, or a runtime error otherwise.

Operator	Type of One Operand	Type of Other Operand	Type of Result
'*', '/', '%'	byte, short, int	byte, short, int	int
	long	any numeric except float or double	long
	float	any numeric except double	float
	double	any numeric	double

continues on next page

Table 1 – continued from previous page

Operator	Type of One Operand	Type of Other Operand	Type of Result
	bigint	bigint	bigint
'**'	any numeric type	any numeric type	double
	bigint	bigint	bigint
'+'	string	converted to a string	string
'+', '-'	byte, short, int	byte, short, int	int
	long	any numeric except float or double	long
	float	any numeric except double	float
	double	any numeric	double
	bigint	bigint	bigint
'<<', '>>'	bigint	bigint	bigint
'<<', '>>', and '>>>'	byte, short, int, float	any numeric	int (see Note 2)
	long, double	any numeric	long (see Note 2)
'<', '<=', '>', '>='	string, string literal	string, string literal	boolean
	boolean	boolean	boolean
	numeric	numeric	boolean
	enum (numeric value)	enum (numeric value)	boolean
	enum (string value)	enum (string value)	boolean
'==', '===', '!=', '!=='	string, string literal	string, string literal	boolean
	boolean	boolean	boolean
	numeric	numeric	boolean
	bigint	bigint	boolean
	char	char	boolean
	enum	enum	boolean
	function	function	boolean
	null, undefined	null, undefined	boolean
'&', '^', '|'	boolean	boolean	boolean
	byte, short, int, float	byte, short, int, float	int
	long, double	long, double	long
	bigint	bigint	bigint
'&&', '||'	boolean	boolean	boolean
	other types, see Note 5	other types, see Note 5	see Note 5

Note

The following applies to the table above.

1. For all operators, except shift operators, the order of operands does not influence the type of result.
2. The type of result of shift operators '<<', '>>', '>>>' with operands of numeric types depends on the type of the 1st (left-hand-side) operand, and not on the type of the second (right-hand-side) operand.
3. Equality operators '==', '===', '!=', and '!==' are defined for any type, but the combinations listed explicitly in the table have specific behaviors.
4. Equality operators for the types that contain `null` or `undefined` are described in *Extended Equality with null or undefined*.
5. Types of operands (other than `boolean`) and the resultant type for logical operators '&&' and '||' are discussed in *Extended Conditional Expressions*.

7.23 Multiplicative Expressions

Multiplicative expressions use *multiplicative operators* `'*'`, `'/'`, and `'%'`.

The syntax of *multiplicative expression* is presented below:

```
multiplicativeExpression:
    expression '*' expression
    | expression '/' expression
    | expression '%' expression
    ;
```

Multiplicative operators group left-to-right.

Type of both operands in a multiplicative operator must be one of the following:

- `bigint`; or
- Convertible (see *Numeric Operator Contexts*) to a numeric type (see *Numeric Types*).

Otherwise, a compile-time error occurs.

No implicit conversion is applied to a multiplicative expression with operands of type `bigint`, and the inferred type is always `bigint`.

This behavior is represented by the following example:

```
// OK, the inferred type of 'i' is ``bigint``
let i = 1n * 2n
// Compile-time error, one operand is not of 'bigint' type
let i = 1n * 2
```

A numeric types conversion (see *Widening Numeric Conversions*) of an expression with operands convertible (see *Numeric Operator Contexts*) to a numeric type is performed on both operands to ensure that the resultant type is the type of the multiplicative expression.

The resultant type of an expression with a numeric operand is inferred from the largest type after promoting `byte` and `short` operands to `int`:

- `double` if any operand is `double`;
- `float` if any operand is `float`, and no operand is `double`;
- `long` if any operand is `long`, and no operand is `double` or `float`;
- `int` if all operands are of type `byte`, `short`, or `int`.

This situation is represented in the following example:

```
// Code below prints true 4 times
let byte1: byte = 1
let byte2: byte = 1
let long1: long = 1
let float1: float = 1
let double1: double = 1

```

(continues on next page)

(continued from previous page)

```
let res_byte = byte1 * byte2  // int
console.log(res_byte instanceof int)

let res_long = byte1 * long1  // long
console.log(res_long instanceof long)

let res_float = byte1 * float1 // float
console.log(res_float instanceof float)

let res_double = byte1 * double1 // double
console.log(res_double instanceof double)
```

7.23.1 Multiplication

The binary operator `'*'` performs multiplication, and returns the product of its operands.

Multiplication is a commutative operation if operand expressions have no side effects.

Bigint multiplication is associative.

Integer multiplication is associative when all operands are of the same type.

Floating-point multiplication is not associative.

Type of a *multiplication expression* with numeric operands is the *largest* type (see *Numeric Types*) of its operands *after* applying *Widening Numeric Conversions*.

If overflow occurs during integer multiplication, then:

- The result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format.
- The sign of the result can be other than the sign of the mathematical product of the two operand values.

A floating-point multiplication result is determined in compliance with the IEEE 754 arithmetic:

- The result is `NaN` if:
  - Either operand is `NaN`;
  - Infinity is multiplied by zero.
- If the result is not `NaN`, then the sign of the result is as follows:
  - Positive, where both operands have the same sign; and
  - Negative, where the operands have different signs.
- If infinity is multiplied by a finite value, then the multiplication results in a signed infinity (the sign is determined by the rule above).
- If neither `NaN` nor infinity is involved, then the exact mathematical product is computed.

  The product is rounded to the nearest value in the chosen value set by using the IEEE 754 *round-to-nearest* mode. The ArkTS programming language requires gradual underflow support as defined by IEEE 754 (see *Floating-Point Types and Operations*).

If the magnitude of the product is too large to represent, then the operation overflows, and the result is an appropriately signed infinity.

The evaluation of a multiplication operator `'*'` never throws an error despite possible overflow, underflow, or loss of information.

7.23.2 Division

The binary operator `'/'` performs division and returns the quotient of its left-hand-side and right-hand-side expressions (`dividend` and `divisor` respectively).

Bigint division rounds toward *0*, i.e., the quotient of bigint operands n and d is the `bigint` value q with the largest possible magnitude that satisfies $|d \cdot q| \leq |n|$.

If the divisor value of the `bigint` division operator is *0n*, then either a compile-time error occurs if detected at compile time, or a runtime error is thrown during execution.

Integer division rounds toward *0*, i.e., the quotient of integer operands n and d, after a numeric types conversion on both (see *Widening Numeric Conversions* for details), is the integer value q with the largest possible magnitude that satisfies $|d \cdot q| \leq |n|$.

Note

The integer value q is:

- Positive, where |n| $\geq$ |d|, and n and d have the same sign; but
- Negative, where |n| $\geq$ |d|, and n and d have opposite signs.

The only special case that does not comply with this rule is where integer overflow occurs. The result equals the dividend if the dividend is a negative integer of the largest possible magnitude for its type, while the divisor is *-1*. No error is thrown in this case despite the overflow.

However, if the divisor value of integer division is detected to be *0* during compilation, then a compile-time error occurs. Otherwise, an `ArithmeticError` is thrown during execution.

The result of a floating-point division is determined in compliance with the IEEE 754 arithmetic:

- The result is `NaN` if:
  - Either operand is NaN;
  - Both operands are infinity; or
  - Both operands are zero.
- If the result is not `NaN`, then the sign of the result is:
  - Positive, where both operands have the same sign; or
  - Negative, where the operands have different signs.
- Division produces a signed infinity (the sign is determined by the rule above) if:
  - Infinity is divided by a finite value; and
  - A nonzero finite value is divided by zero.
- Division produces a signed zero (the sign is determined by the rule above) if:

  - A finite value is divided by infinity; and
  - Zero is divided by any other finite value.

- If neither `NaN` nor infinity is involved, then the exact mathematical quotient is computed.

  If the magnitude of the product is too large to represent, then the operation overflows, and the result is an appropriately signed infinity.

The quotient is rounded to the nearest value in the chosen value set by using the IEEE 754 *round-to-nearest* mode. The ArkTS programming language requires gradual underflow support as defined by IEEE 754 (see *Floating-Point Types and Operations*).

The evaluation of a floating-point division operator `'/'` never throws an error despite possible overflow, underflow, division by zero, or loss of information.

The type of a *division expression* with operands of numeric types is the *largest* numeric type (see *Numeric Types*) of its operands *after* applying *Widening Numeric Conversions*.

7.23.3 Remainder

The binary operator `'%'` yields the remainder of its operands (`dividend` as the left-hand-side, and `divisor` as the right-hand-side operand) from an implied division.

The remainder operator in ArkTS accepts floating-point operands (unlike in C and C++).

A remainder operation $(a\%b)$ on `bigint` operands produces a result that makes $(a/b) * b + (a\%b)$ equal a. No implicit conversion is applied to an operand.

If the divisor value of the `bigint` remainder operator is *0n*, then a runtime error is thrown during execution.

The remainder operation $(a\%b)$ on integer operands produces a result that makes $(a/b) * b + (a\%b)$ equal a. Numeric type conversion on remainder operation is discussed in *Widening Numeric Conversions*. A remainder operation produces a value of type `int` for `byte`, `short`, `int`, and `float` operands, or a value of type `long` for `long` or `double` operands.

This equality holds even in the special case where the dividend is a negative integer of the largest possible magnitude of its type, and the divisor is *-1* (the remainder is then *0*). According to this rule, the result of the remainder operation can only be one of the following:

- Negative if the dividend is negative; or
- Positive if the dividend is positive.

The magnitude of the result is always less than that of the divisor.

If the divisor value of integer remainder operator is detected to be *0* during compilation, then a compile-time error occurs. Otherwise, an `ArithmeticError` is thrown during execution.

The result of a floating-point remainder operation as computed by the operator `'%'` is different than that produced by the remainder operation defined by IEEE 754. The IEEE 754 remainder operation computes the remainder from a rounding division (not a truncating division), and its behavior is different from that of the usual integer remainder operator. On the contrary, ArkTS presumes that the operator `'%'` behaves on floating-point operations in the same manner as the integer remainder operator (comparable to the C library function *fmod*). The standard library (see *Standard Library*) routine `Math.IEEEremainder` can compute the IEEE 754 remainder operation.

The result of a floating-point remainder operation is determined in compliance with the IEEE 754 arithmetic:

- The result is `NaN` if:

  - Either operand is `NaN`;
  - The dividend is infinity;
  - The divisor is zero; or
  - The dividend is infinity, and the divisor is zero.
- If the result is not `NaN`, then the sign of the result is the same as the sign of the dividend.
- The result equals the dividend if:
  - The dividend is finite, and the divisor is infinity; or
  - If the dividend is zero, and the divisor is finite.
- If infinity, zero, or `NaN` are not involved, then the floating-point remainder *r* from the division of the dividend *n* by the divisor *d* is determined by the mathematical relation $r = n - (d \cdot q)$, where *q* is an integer that is only:
  - Negative if n/d is negative, or
  - Positive if n/d is positive.
- The magnitude of *q* is the largest possible without exceeding the magnitude of the true mathematical quotient of *n* and *d*.

The evaluation of the floating-point remainder operator `'%'` never throws an error, even if the right-hand operand is zero. Overflow, underflow, or loss of precision cannot occur.

The type of a *remainder expression* with numeric operands is the *largest* numeric type (see *Numeric Types*) of its operands *after* applying *Widening Numeric Conversions*.

7.24 Exponentiation Expression

Exponentiation expression uses the binary *exponentiation operator* '**'.

The binary operator '**' raises the first operand (base) to the power of the second operand (exponent).

The syntax of an *exponentiation expression* is presented below:

```
exponentiationExpression:
    expression '**' expression
    ;
```

The operator '**' has two variants distinguishable by operand types, i.e., both operands are as follows:

- *Type bigint*; or
- *Numeric Types*, in which case any operand that is not of type `double` is converted to type `double`.

Any other combination of operand types causes a compile-time error.

The result of raising to power *0n* is always *1n*, including the case *0n* ** *0n*.

If the second operand of type `bigint` is negative, then either a compile-time error occurs if detected at compile time, or a runtime error is thrown during execution.

Both variants of the operator '**' are represented in example below:

```
let a: bigint = 2n
let b: double = 2
let c: int = 2
let d: int = 10

let v = a ** 2n // OK 'bigint' ** 'bigint'
let u = a ** 0n // OK 'bigint' ** 'bigint'
let w = a ** -1n // Compile-time error, exponent must be non-negative

let x = a ** c // Compile-time error, 'c' is not 'bigint'
let y = b ** d // 'd' is converted to 'double'
let z = c ** d // both 'c' and 'd' are converted to 'double'
```

The binary operator '**' with *numeric operands* is equivalent to *Math.pow()*. It causes neither a compile-time error, nor a runtime error.

Special cases of the binary operator '**' according to IEEE 754 are represented below.

Note

Since any numeric operand of '**' is implicitly converted to `double`, the term `integer` effectively means `double` with zero in the fractional part (for example *-2.0*) as listed below:

- *x* ** *+/-0* returns *1* even if *x* is *NaN*;
- *+/-0* ** *y* returns *+/-Infinity* if *y* is a negative odd integer;
- *+/-0* ** *-Infinity* returns *+Infinity*;
- *+/-0* ** *+Infinity* returns *+0*;
- *+/-0* ** *y* returns *+0* if *y* is a finite positive odd integer;
- *-1* ** *+Infinity* returns *1*;
- *+1* ** *y* returns *1* for any *y* (even for *NaN*);
- *x, +Infinity* returns *+0* for *-1 < x < 1*;
- *x, +Infinity* returns *+Infinity* for *x < -1 or for *1 < x* (including *+/-Infinity*);
- *x, -Infinity* returns *+Infinity* for *-1 < x < 1*;
- *x, -Infinity* returns *+0* for *x < -1* or for *1 < x* (including +/-Infinity);
- *+Infinity, y* returns *+0* for a number *y < 0*;
- *+Infinity, y* returns *+Infinity* for a number *y > 0*;
- *-Infinity, y* returns *-0* for a finite negative odd integer *y*;
- *-Infinity, y* returns *-Infinity* for a finite positive odd integer *y*;
- *-Infinity, y* returns *+0* if *y* for a finite *y < 0* and not an odd integer;
- *-Infinity, y* returns *+Infinity* for a finite *y > 0* and not an odd integer;
- *+/-0, y* returns *+Infinity* for a finite *y < 0* and not an odd integer;
- *+/-0, y* returns *+0* for a finite *y > 0* and not an odd integer;
- *x, y* returns *NaN* for a finite *x < 0* and a finite non-integer *y*.

7.25 Additive Expressions

Additive expressions use *additive operators* `'+'` and `'-'`.

The syntax of *additive expression* is presented below:

```
additiveExpression:
    expression '+' expression
    | expression '-' expression
    ;
```

Additive operators group left-to-right.

The following rules apply where the operator `'+'` is used:

- If either operand is of type `string`, then the operation is a string concatenation (see *String Concatenation*).
- If both operands are of type `bigint`, then no implicit conversion is applied, and the inferred type is `bigint`.
- In all other cases, type of each operand must be convertible (see *Widening Numeric Conversions*) to a numeric type (see *Numeric Types*).

Otherwise, a compile-time error occurs.

The following rules apply where the operator `'-'` is used:

- Either both operands must be of type `bigint`; or
- Type of each operand of a binary operator must be convertible (see *Widening Numeric Conversions*) to a numeric type (see *Numeric Types*).

Otherwise, a compile-time error occurs.

Type of an *additive expression* with a valid combination of types is determined as follows:

- If any operand is of type `string`, then `string`;
- If both operands are of type `bigint`, then `bigint`;
- If both operands are convertible to a numeric type, then the type inferred after widening operands of numeric types by the rules explained in the example in *Multiplicative Expressions*.

7.25.1 String Concatenation

If one operand of an expression is of type `string`, then the string conversion (see *String Operator Contexts*) is performed on the other operand at runtime to produce a string.

String concatenation produces a reference to a `string` object that is a concatenation of two operand strings. The left-hand-side operand characters precede the right-hand-side operand characters in a newly created string.

If the expression is not a constant expression (see *Constant Expressions*), then a new `string` object is created (see *New Expressions*).

7.25.2 Additive Operators for Numeric Types

A numeric types conversion (see *Widening Numeric Conversions*) performed on a pair of operands ensures that both operands are of a numeric type. If the conversion fails, then a compile-time error occurs.

The binary operator `'+'` performs addition and produces the sum of such operands.

The binary operator `'-'` performs subtraction and produces the difference of two numeric operands.

Type of an additive expression performed on numeric operands is the largest type (see *Numeric Types*) to which operands of that expression are converted (see *Multiplicative Expressions* for an example).

If the promoted type is `int` or `long`, then integer arithmetic is performed. If the promoted type is `float` or `double`, then floating-point arithmetic is performed.

If operand expressions have no side effects, then addition is a commutative operation.

If all operands are of the same type, then integer addition is associative.

Floating-point addition is not associative.

If overflow occurs on an integer addition, then:

- Result is the low-order bits of the mathematical sum as represented in a sufficiently large two's-complement format.
- Sign of the result is opposite to that of the mathematical sum of the operands' values.

The result of a floating-point addition is determined in compliance with the IEEE 754 arithmetic as follows:

- The result is `NaN` if:
  - Either operand is `NaN`; or
  - The operands are two infinities of the opposite signs.
- The sum of two infinities of the same sign is the infinity of that sign.
- The sum of infinity and a finite value equals the infinite operand.
- The sum of two zeros of opposite sign is positive zero.
- The sum of two zeros of the same sign is zero of that sign.
- The sum of zero and a nonzero finite value is equal to the nonzero operand.
- The sum of two nonzero finite values of the same magnitude and opposite sign is positive zero.
- If infinity, zero, or `NaN` are not involved, and the operands have the same sign or different magnitudes, then the exact sum is computed mathematically.

If the magnitude of the sum is too large to represent, then the operation overflows. The result is an appropriately signed infinity.

Otherwise, the sum is rounded to the nearest value within the chosen value set by using the IEEE 754 *round-to-nearest* mode. The ArkTS programming language requires gradual underflow support as defined by IEEE 754 (see *Floating-Point Types and Operations*).

When applied to two numeric type operands (see *Numeric Types*), the binary operator `'-'` performs subtraction, and returns the difference of such operands (`minuend` as left-hand-side, and `subtrahend` as the right-hand-side operand).

The result of *a-b* is always the same as that of *a+(-b)* in both integer and floating-point subtraction.

The subtraction from zero for integer values is the same as negation. However, the subtraction from zero for floating-point operands and negation is *not* the same (if *x* is *+0.0*, then *0.0-x* is *+0.0*; however *-x* is *-0.0*).

The evaluation of a numeric additive operator never throws an error despite possible overflow, underflow, or loss of information.

7.26 Shift Expressions

Shift expressions use *shift operators* `'<<'` (left shift), `'>>'` (signed right shift), and `'>>>'` (unsigned right shift). The value to be shifted is the left-hand-side operand in a shift operator, and the right-hand-side operand specifies the shift distance.

The syntax of *shift expression* is presented below:

```
shiftExpression:
    expression '<<' expression
    | expression '>>' expression
    | expression '>>>' expression
    ;
```

Shift operators group left-to-right.

Both operands of a *shift expression* must be of numeric types or type `bigint`.

If the type of one or both operands is `double` or `float`, then the operand or operands are truncated first to `long` or `int`, respectively. If the type of the left-hand-side operand is `byte` or `short`, then the operand is converted to `int`. If both operands are of type `bigint`, then no conversion is required. A compile-time error occurs if one operand is type `bigint`, and the other one is a numeric type. Also, a compile-time error occurs if `'>>>'` (unsigned right shift) is applied to operands of type `bigint`.

The result of a *shift expression* is of the type to which its first operand converted.

If the left-hand-side operand is of the promoted type `int`, then only five lowest-order bits of the right-hand-side operand specify the shift distance (as if using a bitwise logical AND operator `'&'` with the mask value *0x1f* or *0b11111* on the right-hand-side operand). Thus, it is always within the inclusive range of *0* through *31*.

If the left-hand-side operand is of the promoted type `long`, then only six lowest-order bits of the right-hand-side operand specify the shift distance (as if using a bitwise logical AND operator `'&'` with the mask value *0x3f* or *0b111111* the right-hand-side operand). Thus, it is always within the inclusive range of *0* through *63*.

Shift operations are performed on the two's-complement integer representation of the value of the left-hand-side operand at runtime.

The value of *n* << *s* is *n* left-shifted by *s* bit positions. It is equivalent to multiplication by two to the power *s* even in case of an overflow.

The value of *n* >> *s* is *n* right-shifted by *s* bit positions with sign-extension. The resultant value is $floor(n/2s)$. If *n* is non-negative, then it is equivalent to truncating integer division (as computed by the integer division operator by 2 to the power *s*).

The value of *n* >>> *s* is *n* right-shifted by *s* bit positions with zero-extension, where:

- If *n* is positive, then the result is the same as that of *n* >> *s*.
- If *n* is negative, and type of the left-hand-side operand is `int`, then the result is equal to that of the expression (*n* >> *s*) + `(2 << ~ s)`.
- If *n* is negative, and type of the left-hand-side operand is `long`, then the result is equal to that of the expression (*n* >> *s*) + `((2 as long) << ~ s)`.

7.27 Relational Expressions

Relational expressions use *relational operators* '<', '>', '<=', and '>='.

The syntax of *relational expression* is presented below:

```
relationalExpression:
    expression '<' expression
    | expression '>' expression
    | expression '<=' expression
    | expression '>=' expression
    ;
```

Relational operators group left-to-right.

A relational expression is always of type `boolean`.

Relational expressions can be of the kinds described below. The kind of a relational expression depends on types of operands. It is a compile-time error if at least one type of operands is different from types described below.

7.27.1 Numeric Relational Operators

Type of each operand in a `numeric relational operator` must be of a numeric type (see *Numeric Types*, *Numeric Conversions for Relational and Equality Operands*). Otherwise, a compile-time error occurs.

Depending on the converted type of operands, a comparison is performed as follows:

- Signed integer comparison, if the converted operand type is `int` or `long`.
- Floating-point comparison, if the converted operand type is `float` or `double`.

The comparison of floating-point values drawn from any value set must be accurate.

A floating-point comparison must be performed in accordance with the IEEE 754 standard specification as follows:

- The result of a floating-point comparison is false if either operand is `NaN`.
- All values other than `NaN` must be ordered with the following:
  - Negative infinity less than all finite values; and
  - Positive infinity greater than all finite values.
- Positive zero equals negative zero.

Based on the presumption above, the following rules apply to integer operands, floating-point operands other than `NaN`:

- The value produced by the operator '<' is `true` if the value of the left-hand-side operand is less than that of the right-hand-side operand. Otherwise, the value is `false`.
- The value produced by the operator '<=' is `true` if the value of the left-hand-side operand is less than or equal to that of the right-hand-side operand. Otherwise, the value is `false`.
- The value produced by the operator '>' is `true` if the value of the left-hand-side operand is greater than that of the right-hand-side operand. Otherwise, the value is `false`.
- The value produced by the operator '>=' is `true` if the value of the left-hand-side operand is greater than or equal to that of the right-hand-side operand. Otherwise, the value is `false`.

The behavior of comparison of `numeric` type operands is represented in the example below:

```
// integer comparisons
console.log( 1 < -3) // false
console.log(-1 as long >= -1 as short) // true

// floating-point comparisons
console.log(1 <= 1.0f)  // true
console.log(2 <= 1.0)   // false
```

7.27.2 Bigint Relational Operators

Relational operators can be applied to `bigint` values (see *Type bigint*) if:

- both operands are of `bigint` type; or
- one operand is of `bigint` type and other is of a numeric type.

In the second case, a comparison is performed as follows:

- If a numeric type operand is of an integer type, it is converted to `bigint` type and `bigint` comparison is performed;
- If a numeric type operand is of `double` type, the `bigint` operand is converted to `double` and then floating-point comparison is performed;
- If a numeric type operand is of `float` type, both operands are converted to `double` and then floating-point comparison is performed.

The behavior of comparison of `numeric` type operands and `bigint` type operands is represented in the example below:

```
let b = 2n

// bigint against bigint
console.log(b < 3n)  // true
console.log(b >= 3n) // false

// bigint against an integer type
console.log(b < 3)         // true, '2' is converted implicitly to bigint
console.log(b < BigInt(3)) // true, the same with explicit conversion

// bigint against double
console.log(b < 3.0)               // true, 'b' is converted to double
console.log(b.doubleValue() < 3.0) // the same with explicit conversion

// bigint against float
const f = 3.f
console.log(b < f) // true, 'b' and 'f' are converted to bigint
```

7.27.3 String Relational Operators

Results of all string comparisons are defined as follows:

- Operator '<' delivers `true` if the string value of the left-hand-side operand is lexicographically less than the string value of the right-hand-side operand, or `false` otherwise.
- Operator '<=' delivers `true` if the string value of the left-hand-side operand is lexicographically less than or equal to the string value of the right-hand-side operand, or `false` otherwise.
- Operator '>' delivers `true` if the string value of the left-hand-side operand is lexicographically greater than the string value of the right-hand-side operand, or `false` otherwise.
- Operator '>=' delivers `true` if the string value of the left-hand-side operand is lexicographically greater than or equal to the string value of the right-hand operand, or `false` otherwise.

7.27.4 Boolean Relational Operators

Results of all boolean comparisons are defined as follows:

- Operator '<' delivers `true` if the left-hand-side operand is `false`, and the right-hand-side operand is true, or `false` otherwise.
- Operator '<=' delivers:
  - `true` when both operands are `true`, or the left-hand-side operand is `false` for any right-hand value;
  - `false` when the left-hand-side operand is `true`, and the right-hand-side operand is `false`.
- Operator '>' delivers `true` if the left-hand-side operand is `true`, and the right-hand-side operand is `false`, or `false` otherwise.
- Operator '>=' delivers:
  - `true` when both operands are `false`, or the left-hand-side operand is `true` for any right-hand-side value;
  - `false` when the left-hand-side operand is `false`, and the right-hand-side operand is `true`.

7.27.5 `char` Relational Operators

See *char Operations*.

7.27.6 Enumeration Relational Operators

If both operands are of the same enumeration type (see *Enumerations*), then *Numeric Relational Operators* or *String Relational Operators* are used depending on the type of enumeration base type. Otherwise, a compile-time error occurs.

7.28 Equality Expressions

Equality expressions use *equality operators* `'=='`, `'==='`, `'!='`, and `'!=='`.

The syntax of *equality expression* is presented below:

```
equalityExpression:
    expression ('==' | '===' | '!=' | '!==') expression
    ;
```

Equality operators group left-to-right. Equality operators are commutative if operand expressions cause no side effects.

Similarly to relational operators, equality operators return `true` or `false`. Equality operators have lower precedence than relational operators, for example, $a < b == c < d$ is *true* when both $a < b$ and $c < d$ are `true`.

Any equality expression is of type `boolean`.

The result produced by `a != b` and `!(a == b)` is the same in all cases. The result produced by `a !== b` and `!(a === b)` is the same.

The result of the operators `'=='` and `'==='` is the same in all cases except when comparing the values `null` and `undefined` (see *Extended Equality with null or undefined*).

A comparison that uses the operators `'=='` and `'==='` is evaluated to `true` if:

- Both operands are of *Type boolean* and have the same value;
- Both operands are of *Type string* or string literal type (see *String Literal Types*) and have the same contents;
- Both operands are of *Type bigint* and have the same value;
- Both operands are of *Numeric Types* and have the same value except `NaN` (see *Numeric Equality Operators* for detail) after a numeric conversion (see *Widening Numeric Conversions*, *Numeric Conversions for Relational and Equality Operands*);
- Both operands are of *Type char* and have the same value, i.e., both operands represent the same 16-bit code unit (see *char Operations*);
- One operand is of *Type char*, other is of a numeric type and operand have the same numeric value (see *char Operations* and *char Conversions for Relational and Equality Operands*);
- Both operands are of enumeration types (see *Enumerations*) and have the same numeric value or the same string value;
- Function references refer to the same functional object (see *Function Type Equality Operators* for detail).
- Both operands are of the same type and refer to the same object.

An evaluation of equality expressions always uses the actual types of operands as in the example below:

```
function equ(a: Object, b: Object): boolean {
    return a == b
}

equ(1, 1) // true, values are compared
equ(1, 2) // false, value are compared

equ("aa", "aa") // true, string contexts are compared
equ(1, "aa")    // false, not compatible types

interface I1 {}
```

(continues on next page)

(continued from previous page)

```
interface I2 {}

function equ1 (i1: I1, i2: I2) {
   return i1 == i2 // to be resolved during program execution
}
class A implements I1, I2 {}
const a = new A
equ1 (a, a) // true, the same values
```

An equality with values of two union types is represented in the example below:

```
function f1(x: number | string, y: boolean | undefined): boolean {
    return x == y // Compile-time warning, always evaluates to false
}

function f2(x: number | string, y: boolean | "abc"): boolean {
    // OK, can be evaluated as true
    return x == y
}
```

If an *equality expression* is known at compile time to always evaluate to `false` or `true` at runtime, then a compile-time warning is issued as follows:

```
5 == 5  // Compile-time warning, always true
5 != 5  // Compile-time warning, always false
5 === 5 // Compile-time warning, always true
5 !== 5 // Compile-time warning, always false

function  foo(b: boolean | undefined) {
    let n: number | boolean = 1
    b == n // Compile-time warning, expression is always false
           // (type of `n` is `number` and does not overlap with type of `b`)
    n == 1 // Compile-time warning, expression is
           // always true because `n` initialized with 1 and never modified
}

class B {
    f(): B|undefined { return undefined }
}
class D extends B {
    f(): D { return this }
}

function bar(c: B) {
    if (c instanceof D) {
        c.f() == undefined // warning, expression is always false
    }
}
```

If two operands are of different enumeration types, then a compile-time warning occurs as follows:

```
enum E1 {A, B}
enum E2 {C, D}
```

(continues on next page)

(continued from previous page)

```
3
4  function test (e1: E1, e2: E2) {
5     e1 == e2 // Compile-time warning, appears to be unintentional
6  }
```

7.28.1 Numeric Equality Operators

Type of each operand in a `numeric equality operator` must be convertible to a numeric type (see *Numeric Types*) or to a `bigint` type (see *Type bigint*) as described in *Numeric Conversions for Relational and Equality Operands*. Otherwise, a compile-time error occurs.

A widening conversion can occur (see *Widening Numeric Conversions*) if type of one operand is smaller than type of the other operand (see *Numeric Types*).

If the converted type of the operands is `int` or `long`, then an integer equality test is performed.

If the converted type is `float` or `double`, then a floating-point equality test is performed.

The floating-point equality test must be performed in accordance with the following IEEE 754 standard rules:

- The result of `'=='` or `'==='` is `false` but the result of `'!='` is `true` if either operand is `NaN`.

  The test `x != x` or `x !== x` is `true` only if *x* is `NaN`.

- Positive zero equals negative zero.

- Equality operators consider two distinct floating-point values unequal in any other situation.

  For example, if one value represents positive infinity, and the other represents negative infinity, then each compares equal to itself and unequal to all other values.

Based on the above presumptions, the following rules apply to integer operands or floating-point operands other than `NaN`:

- If the value of the left-hand-side operand is equal to that of the right-hand-side operand, then the operator `'=='` or `'==='` produces the value `true`. Otherwise, the result is `false`.

- If the value of the left-hand-side operand is not equal to that of the right-hand-side operand, then the operator `'!='` or `'!=='` produces the value `true`. Otherwise, the result is `false`.

```
1  5 == 5 // true
2  5 != 5 // false
3
4  5 === 5 // true
5
6  5 == new Number(5) // true
7  5 === new Number(5) // true
8
9  5 == 5.0 // true
```

If *both* operands are of type `bigint` and have the same value, then the operators `'=='` or `'==='` produce the value `true`. Otherwise, the result is `false`. The result produced by `a != b` and `a !== b` is the same as the result of `!(a == b)` and `!(a === b)`, respectively.

If one operand is of type `bigint`, and the other is of a numeric type, then the result is `false`.

7.28.2 Function Type Equality Operators

If both operands refer to the same function object, then the comparison returns `true`. When comparing method references, not only the same method must be used, but also its bounded instances must be equal.

```
function foo() {}
function bar() {}
function goo(p: number) {}

foo == foo // true, same function object
foo == bar // false, different function objects
foo == goo // false, different function objects

class A {
   method() {}
   static method() {}
   foo () {}
}
const a = new A
a.method == a.method // true, same function object
A.method == A.method // true, same function object

const aa = new A
a.method == aa.method /* false, different function objects
     as 'a' and 'aa' are different bounded objects */
a.method == a.foo // false, different function objects
```

7.28.3 Extended Equality with `null` or `undefined`

ArkTS provides extended semantics for equalities with `null` and `undefined` to ensure better alignment with TypeScript.

If one operand in an equality expression is `null`, and other is `undefined`, then the operator `'!='` returns `true`, and the operator `'!=='` returns `false`:

```
function foo(x: Object | null, y: Object | null | undefined) {
    console.log(x == y, x === y)
}

foo(null, undefined) // output: true, false
foo(null, null)      // output: true, true
```

Comparison the values `null` and `undefined` directly is also allowed:

```
console.log(null == undefined)  // output: true
console.log(null === undefined) // output: false
```

7.29 Bitwise and Logical Expressions

The *bitwise operators* and *logical operators* are as follows:

- AND operator `'&'`;
- Exclusive OR operator `'^'`; and
- Inclusive OR operator `'|'`.

The syntax of *bitwise and logical expression* is presented below:

```
bitwiseAndLogicalExpression:
    expression '&' expression
    | expression '^' expression
    | expression '|' expression
    ;
```

These operators have different precedence. The operator `'&'` has the highest, while `'|'` has the lowest precedence.

Operators group left-to-right. Each operator is commutative if the operand expressions have no side effects, and associative.

The bitwise and logical operators can compare two operands of a numeric type, or two operands of the `boolean` type. Otherwise, a compile-time error occurs.

7.29.1 Integer Bitwise Operators

Integer bitwise operators are `'&'`, `'^'`, and `'|'` applied to operands of numeric types or type `bigint`.

If the type of one or both operands is `double` or `float`, then the operand or operands are truncated first to the appropriate integer type. If the type of any operand is `byte` or `short`, then the operand is converted to `int`. If operands are of different integer types, then the operand of a smaller type is converted to a larger type (see *Numeric Types*) by using *Widening Numeric Conversions*. If both operands are of type `bigint`, then no conversion is required. A compile-time error occurs if one operand of type `bigint`, and the other operand is of a numeric type.

The resultant type of the bitwise operator is the type of its operands.

The resultant value of `'&'` is the bitwise AND of the operand values.

The resultant value of `'^'` is the bitwise exclusive OR of the operand values.

The resultant value of `'|'` is the bitwise inclusive OR of the operand values.

7.29.2 Boolean Logical Operators

Boolean logical operators are `'&'`, `'^'`, and `'|'` applied to operands of type `boolean`.

If both operand values are `true`, then the resultant value of `'&'` is `true`. Otherwise, the result is `false`.

If the operand values are different, then the resultant value of `'^'` is `true`. Otherwise, the result is `false`.

If both operand values are `false`, then the resultant value of `'|'` is `false`. Otherwise, the result is `true`.

Thus, *boolean logical expression* is of the boolean type.

7.30 Conditional-And Expression

The *conditional-and* operator `'&&'` is similar to `'&'` (see *Bitwise and Logical Expressions*) but evaluates its right-hand-side operand only if the value of the left-hand-side operand is `true`.

The computation results of `'&&'` and `'&'` on `boolean` operands are the same. The right-hand-side operand of `'&&'` is not necessarily evaluated.

The syntax of *conditional-and expression* is presented below:

```
conditionalAndExpression:
    expression '&&' expression
    ;
```

A *conditional-and* operator groups left-to-right.

A *conditional-and* operator is fully associative as regards both the result value and side effects (i.e., the evaluations of the expressions *((a) && (b)) && (c)* and *(a) && ((b) && (c))* produce the same result, and the same side effects occur in the same order for any *a*, *b*, and *c*).

A *conditional-and* expression is always of type `boolean` except the extended semantics (see *Extended Conditional Expressions*).

Each operand of the *conditional-and* operator must be of type `boolean`, or of a type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

The left-hand-side operand expression is first evaluated at runtime.

If the resultant value is `false`, then the value of the *conditional-and* expression is `false`. The evaluation of the right-hand-side operand expression is omitted.

If the value of the left-hand-side operand is `true`, then the right-hand-side expression is evaluated. The resultant value is the value of the *conditional-and* expression.

7.31 Conditional-Or Expression

The *conditional-or* operator `'||'` is similar to `'|'` (see *Integer Bitwise Operators*) but evaluates its right-hand-side operand only if the value of its left-hand-side operand is `false`.

The syntax of *conditional-or expression* is presented below:

```
conditionalOrExpression:
    expression '||' expression
    ;
```

A *conditional-or* operator groups left-to-right.

A *conditional-or* operator is fully associative as regards both the result value and side effects (i.e., the evaluations of the expressions *((a) || (b)) || (c)* and *(a) || ((b) || (c))* produce the same result, and the same side effects occur in the same order for any *a*, *b*, and *c*).

A *conditional-or* expression is always of type `boolean` except the extended semantics (see *Extended Conditional Expressions*).

Each operand of the *conditional-or* operator must be of type `boolean` or type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

The left-hand-side operand expression is first evaluated at runtime.

If the resultant value is `true`, then the value of the *conditional-or* expression is `true`, and the evaluation of the right-hand-side operand expression is omitted.

If the resultant value is `false`, then the right-hand-side expression is evaluated. The resultant value is the value of the *conditional-or* expression.

The computation results of `'||'` and `'|'` on `boolean` operands are the same, but the right-hand-side operand in `'||'` cannot be evaluated.

7.32 Assignment

All *assignment operators* group right-to-left (i.e., $a = b = c$ means $a = (b = c)$. The value of *c* is thus assigned to *b*, and then the value of *b* to *a*).

The syntax of *assignment expression* is presented below:

```
assignmentExpression
    : lhsExpression assignmentOperator rhsExpression
    | destructuringAssignment
    ;

assignmentOperator
    : '='
    | '+='  | '-='  | '*='   | '/='  | '%=' | '**='
    | '<<=' | '>>=' | '>>>='
    | '&='  | '|='  | '^=' | '&&=' | '||='
    | '??='
    ;

lhsExpression:
    expression
    ;

rhsExpression:
    expression
    ;
```

The first operand in an assignment operator represented by *lhsExpression* must be a *left-hand-side expression* (see *Left-Hand-Side Expressions*). This first operand denotes a variable.

The `destructuringAssignment` expression is discussed in *Destructuring Assignment*.

Type of the variable is the type of the assignment expression.

The result of the *assignment expression* at runtime is not a variable itself but the value of a variable after the assignment.

7.32.1 Simple Assignment Operator

The form of a simple assignment expression is `lhsExpression = rhsExpression`.

A compile-time error occurs in the following situations:

- Type of *rhsExpression* is not assignable (see *Assignability*) to the type of a variable referred by the *lhsExpression*; or
- Type of *lhsExpression* is one of the following:
  - `readonly` array (see *Readonly Parameters*), while the converted type of *rhsExpression* is a non-`readonly` array;
  - `readonly` tuple (see *Readonly Parameters*), while the converted type of *rhsExpression* is a non-`readonly` tuple.

Otherwise, the assignment expression is evaluated at runtime in one of the following ways:

1. If *lhsExpression* is a field access expression in the form `e.f` (see *Field Access Expression*), where *e* is an expression and *f* is the name of the field, then:
   1. Expression *e* is evaluated: if the evaluation of *e* completes abruptly, then so does the assignment expression.
   2. *rhsExpression* is evaluated: if the evaluation completes abruptly, then so does the assignment expression.
   3. After both evaluations complete normally, the value of *rhsExpression* is converted to the type of the field *f*, and the result of the conversion is assigned to the field *f*.
2. If the *lhsExpression* is an array reference expression (see *Array Indexing Expression*) then:
   1. Array reference subexpression of *lhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* and the index subexpression are not evaluated, and no assignment occurs.
   2. If the evaluation completes normally, then the index subexpression of *lhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
   3. If the evaluation completes normally, then *rhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression, and no assignment occurs.
   4. If the evaluation completes normally, but the value of the index subexpression is less than zero, or greater than, or equal to the *length* of the array, then `RangeError` is thrown, and no assignment occurs.
   5. If *lhsExpression* denotes indexing of *fixed-size array*, and the type of *rhsExpression* is not a subtype of array element type, then *ArrayStoreError* is thrown, and no assignment occurs.
   6. Otherwise, the value of the index subexpression is used to select an element of the array referred to by the value of the array reference subexpression and the value of *rhsExpression* is converted to the type of the array element. In that case, the result of the conversion is assigned to the array element.
3. If *lhsExpression* is a record access expression (see *Record Indexing Expression*) then:

1. Object reference subexpression of *lhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* and the index subexpression are not evaluated, and no assignment occurs.
2. If the evaluation completes normally, the index subexpression of *lhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
3. If the evaluation completes normally, *rhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
4. Otherwise, the value of the index subexpression is used as the `key`, and the value of *rhsExpression* converted to the type of the record value is used as the `value`. In that case, the assignment results in storing the key-value pair in the record instance.

If none of the above is true, then the following three steps are performed:

1. *lhsExpression* is evaluated to produce a variable. If the evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
2. If the evaluation completes normally, then *rhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
3. If that evaluation completes normally, then the value of *rhsExpression* is converted to the type of the left-hand-side variable. In that case, the result of the conversion is assigned to the variable.

Assignment expressions for different kinds of *lhsExpression* are represented in the example below:

```
// Case 1: field access lhsExpression
class A { f: double }
new A().f = 1

// Case 2: array indexing lhsExpression
let b: double[] = [1, 1, 1 ]
b[1] = 2

// Case 3: record indexing lhsExpression
type Keys = 'key1' | 'key2' | 'key3'
let x: Record<Keys, number> = { 'key1': 1, 'key2': 2, 'key3': 3 }
x['key2'] = 8

// None of {field access|array indexing|record indexing}
let c: double
c = 1
c += 2
```

7.32.2 Compound Assignment Operators

A compound assignment expression in the form `lhsExpression op= rhsExpression` is equivalent to `lhsExpression = ((lhsExpression) op (rhsExpression)) as T`, where `T` is the type of *lhsExpression*, except that *lhsExpression* is evaluated only once.

While the nullish-coalescing assignment (`??=`) only evaluates the right operand, and assigns to the left operand if the left operand is `null` or `undefined`.

An assignment expression can be evaluated at runtime in one of the following ways:

1. Where *lhsExpression* is not an indexing expression:
   - *lhsExpression* is evaluated to produce a variable. If the evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
   - If the evaluation completes normally, then the value of *lhsExpression* is saved, and *rhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
   - If the evaluation completes normally, then the saved value of the left-hand-side variable, and the value of *rhsExpression* are used to perform the binary operation as indicated by the compound assignment operator. If the operation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
   - If the evaluation completes normally, then the result of the binary operation converts to the type of the left-hand-side variable. The result of such conversion is stored into the variable.
2. Where *lhsExpression* is an array reference expression (see *Array Indexing Expression*), then:
   - Array reference subexpression of *lhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, the index subexpression, and *rhsExpression* are not evaluated, and no assignment occurs.
   - If the evaluation completes normally, then the index subexpression of *lhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
   - If the evaluation completes normally, the value of the array reference subexpression refers to an array, and the value of the index subexpression is less than zero, greater than, or equal to the *length* of the array, then `RangeError` is thrown. In that case, no assignment occurs.
   - If the evaluation completes normally, then the value of the index subexpression is used to select an array element referred to by the value of the array reference subexpression. The value of this element is saved, and then *rhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
   - If the evaluation completes normally, consideration must be given to the saved value of the array element selected in the previous step. While this element is a variable of type `S`, and `T` is type of *lhsExpression* of the assignment operator determined at compile time:
     - If `T` is a predefined value type, then `S` is the same as `T`.

       The saved value of the array element, and the value of *rhsExpression* are used to perform the binary operation of the compound assignment operator.

       If this operation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.

       If this evaluation completes normally, then the result of the binary operation converts to the type of the selected array element. The result of the conversion is stored into the array element.
     - If `T` is a reference type, then it must be `string`.

       `S` must also be a `string` because the class `string` is the *final* class. The saved value of the array element, and the value of *rhsExpression* are used to perform the binary operation (string concatenation) of the compound assignment operator `'+='`. If this operation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
     - If the evaluation completes normally, then the `string` result of the binary operation is stored into the array element.

3. If *lhsExpression* is a record access expression (see *Record Indexing Expression*):

    - The object reference subexpression of *lhsExpression* is evaluated. If this evaluation completes abruptly, then so does the assignment expression. In that case, the index subexpression and *rhsExpression* are not evaluated, and no assignment occurs.
    - If this evaluation completes normally, then the index subexpression of *lhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, *rhsExpression* is not evaluated, and no assignment occurs.
    - If this evaluation completes normally, the value of the object reference subexpression and the value of index subexpression are saved, then *rhsExpression* is evaluated. If the evaluation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
    - If this evaluation completes normally, the saved values of the object reference subexpression and index subexpression (as the *key*) are used to get the *value* that is mapped to the *key* (see *Record Indexing Expression*), then this *value* and the value of *rhsExpression* are used to perform the binary operation as indicated by the compound assignment operator. If the operation completes abruptly, then so does the assignment expression. In that case, no assignment occurs.
    - If the evaluation completes normally, then the result of the binary operation is stored as the key-value pair in the record instance (as in *Simple Assignment Operator*).

7.32.3 Left-Hand-Side Expressions

A *left-hand-side expression* is an *expression* that refers to one of the following:

- Variable (see *Named Reference*);
- Parameter, except any parameter named `this` (see *Named Reference*);
- Field or setter resultant from a field access (see *Field Access Expression*); or
- Array or record element access (see *Indexing Expressions*).

Otherwise, a compile-time error occurs.

A compile-time error occurs if an *expression*:

- Contains the chaining operator `'?.'` (see *Chaining Operator*); or
- Refers to a readonly entity.

7.33 Ternary Conditional Expressions

The ternary conditional expression `condition?whenTrue:whenFalse` uses the boolean value of the first expression (`condition`) to decide which of other two expressions to evaluate:

```
ternaryConditionalExpression:
    expression '?' expression ':' expression
    ;
```

The ternary conditional operator groups right-to-left (i.e., the meaning of $a?b : c?d : e?f : g$ and $a?b : (c?d : (e?f : g))$ is the same).

The ternary conditional operator `condition?whenTrue:whenFalse` consists of three operand expressions with the separators `'?'` between the first and the second expression, and `':'` between the second and the third expression.

Type of an expression `condition?whenTrue:whenFalse` is determined as follows:

- If the value of `condition` is evaluated as `true` at compile time, then the expression has the type of `whenTrue`.
- If the value of `condition` is evaluated as `false` at compile time, then the expression has the type of `whenFalse`.
- If the value of `condition` is unknown at compile time, then the expression type is a union of types `whenTrue` and `whenFalse` further normalized in accordance with the process discussed in *Union Types Normalization*.

The following steps are performed as the evaluation of a ternary conditional expression occurs at runtime:

1. The first operand (`condition`) of a ternary conditional expression is evaluated first.
2. If the value of the first operand is `true`, then the second operand expression (`whenTrue`) is evaluated. Otherwise, the third operand expression (`whenFalse`) is evaluated. The result of successful evaluation is the result of the ternary conditional expression.

The examples below represent different scenarios with standalone expressions:

```
class A {}
class B extends A {}

// Assuming value of `condition` is unknown at compile time
condition ? new A() : new B() // A | B => A

condition ? 5 : 6             // int

condition ? "5" : 6           // "5" | int
true      ? "5" : 6           // "5"
false     ? "5" : 6           // int
```

7.34 String Interpolation Expressions

'*String interpolation expression*' is a multiline string literal, i.e., a string literal delimited with backticks (see *Multiline String Literal* for detail) that contains at least one *embedded expression*.

The syntax of *string interpolation expression* is presented below:

```
stringInterpolation:
    '``' (BacktricksContentCharacter | embeddedExpression)* '``'
    ;

embeddedExpression:
    '${' expression '}'
    ;
```

An '*embedded expression*' is an expression specified inside curly braces preceded by the *dollar sign* `'$'`. A string interpolation expression is of type `string` (see *Type string*).

When evaluating a *string interpolation expression*, the result of each embedded expression substitutes that embedded expression. An embedded expression must be of type `string`. Otherwise, the implicit conversion to `string` takes place in the same way as with the string concatenation operator (see *String Concatenation*):

```
let a = 2
let b = 2
console.log(`The result of ${a} * ${b} is ${a * b}`)
// prints: The result of 2 * 2 is 4
```

The string concatenation operator can be used to rewrite the above example as follows:

```
let a = 2
let b = 2
console.log("The result of " + a + " * " + b + " is " + a * b)
```

An embedded expression can contain nested multiline strings.

7.35 Lambda Expressions

Lambda expression fully defines an instance of a function type (see *Function Types*) by providing optional annotation usage (see *Using Annotations*), optional `async` mark (see *Asynchronous Lambdas*), mandatory lambda signature, and its body. The declaration of *lambda expression* is generally similar to that of a function declaration (see *Function Declarations*), except that a lambda expression has no function name specified, and can have types of parameters omitted.

The syntax of *lambda expression* is presented below:

```
lambdaExpression:
    annotationUsage? 'async'? lambdaSignature '=>' lambdaBody
    ;

lambdaBody:
    expression | block
    ;

lambdaSignature:
    '(' lambdaParameterList? ')' returnType?
    | identifier
    ;

lambdaParameterList:
    lambdaParameter (',' lambdaParameter)* (',' restParameter)? ','?
    | restParameter ','?
    ;

lambdaParameter:
    annotationUsage? (lambdaRequiredParameter | lambdaOptionalParameter)
    ;

lambdaRequiredParameter:
```

(continues on next page)

(continued from previous page)

```
    identifier (':' type)?
    ;

lambdaOptionalParameter:
    identifier '?' (':' type)?
    ;

lambdaRestParameter:
    '...' lambdaRequiredParameter
    ;
```

The usage of annotations is represented in the examples below and further discussed in *Using Annotations*:

```
(x: number): number => { return Math.sin(x) } // block as lambda body
(x: number) => Math.sin(x)                    // expression as lambda body
e => e                                        // shortest form of lambda
```

A *lambda expression* evaluation creates an instance of a function type (see *Function Types*) as described in detail in *Runtime Evaluation of Lambda Expressions*.

7.35.1 Lambda Signature

Similarly to function declarations (see *Function Declarations*), a *lambda signature* is composed of formal parameters and optional return types. Type inference (see *Type Inference*) allows to omit type annotations of formal parameters in some cases.

```
// Lambda expressions has no type annotations

const func: (p: Object) => Object = e => e

function foo<T>(p: (p: T) => T) {}
foo <Object> (e => e)
```

The specification of scope is discussed in *Scopes*, and shadowing details of formal parameter declarations in *Shadowing by Parameter*.

A compile-time error occurs if:

- Lambda expression declares two formal parameters with the same name.
- Formal parameter contains no type provided, and type cannot be derived by *Type Inference*.

7.35.2 Lambda Body

Lambda body can be a single expression or a block (see *Block*). Similarly to the body of a method or a function, a lambda body describes the code to be executed when a lambda expression call occurs (see *Function Call Expression*).

The meanings of names, and of the keywords `this` and `super` (along with the accessibility of the referred declarations) are the same as in the surrounding context. However, lambda parameters introduce new names.

If any local variable or formal parameter of the surrounding context is used but not declared in a lambda body, then the local variable or formal parameter is *captured* by the lambda.

If an instance member of the surrounding type is used in the lambda body defined in a method, then `this` is *captured* by the lambda.

A compile-time error occurs if a local variable is used in a lambda body but is neither declared in nor assigned before it.

If a *lambda body* is a single `expression`, then it is handled as follows:

- If the expression is a *call expression* with return type `void`, then the body is equivalent to the block: `{ expression }`.
- Otherwise, the body is equivalent to the block: `{ return expression }`.

If *lambda signature* return type is neither `void` (see *Type void or undefined*) nor `never` (see *Type never*), and the execution path of the lambda body has neither a return statement (see *return Statements*) nor a single expression as a body, then a compile-time error occurs.

7.35.3 Lambda Expression Type

Lambda expression type is a function type (see *Function Types*) that has the following:

- Lambda parameters (if any) as parameters of the function type; and
- Lambda return type as the return type of the function type.

Note

Lambda return type can be inferred from the *lambda body* and thus the return type can be dropped off.

```
const lambda = () => { return 123 }  // Type of the lambda is () => int
const int_var: int = lambda()
```

7.35.4 Runtime Evaluation of Lambda Expressions

The evaluation of a lambda expression itself never causes the execution of the lambda body. If completing normally at runtime, the evaluation of a lambda expression produces a new instance of a function type (see *Function Types*) that corresponds to the lambda signature. In that case, it is similar to the evaluation of a class instance creation expression (see *New Expressions*).

If the available space is not sufficient for a new instance to be created, then the evaluation of the lambda expression completes abruptly, and `OutOfMemoryError` is thrown.

Every time a lambda expression is evaluated, the outer variables referred to by the lambda expression are captured as follows:

```
function foo() {
   let y: int = 1
   let x = () => { return y+1 } // 'y' is *captured*.
   console.log(x())             // Output: 2
}
```

The captured variable is not a copy of the original variable. If the value of the variable captured by the lambda changes, then the original variable is implied to change:

```
function foo() {
  let y: int = 1
  let x = () => { y++ } // 'y' is *captured*.
  console.log(y) // Output: 1
  x()
  console.log(y) // Output: 2
}
```

Capturing within the function scope is highlighted by the following example:

```
function capturingFunction() { // Function scope
  let v: number = 0 // A captured variable
  return  (p: number) => {
      console.log ("Previous value: ", v, " new value: ", p)
      v = p
  }
}

const func1 = capturingFunction ()
const func2 = capturingFunction ()
// Note: func1 and func2 are two different function type instances

func1(11) // Previous value: 0 new value: 11
func2(22) // Previous value: 0 new value: 22
func1(33) // Previous value: 11 new value: 33
func2(44) // Previous value: 22 new value: 44
/* Note:
      func1 calls work with their own version of variable 'v'
      func2 calls work with their own version of variable 'v'
*/
```

Capturing within the loop scope is highlighted by the following example:

```
const l = () => {}
const storage = [l, l, l, l, l]  // fill array with some lambdas

for (let index = 0; index < 5; index++) {
   storage [index] = () => { console.log ("Index ", index) }
   // Every lambda captures loop index variable
}
```

(continues on next page)

(continued from previous page)

```
  for (let index = 0; index < 5; index++) {
    storage[index]() // Captured indices printed
  }
```

7.36 Constant Expressions

Constant expressions are expressions with values that can be evaluated at compile time. If the evaluation *completes abruptly*, then a compile-time error occurs.

The syntax of *constant expression* is presented below:

```
constantExpression:
    expression
    ;
```

A *constant expression* can be either of a value type (see *Value Types*) or of type `string`, while being composed only of the following:

- Literals of a predefined value types, and literals of type `string` (see *Literals*);
- Simple names that refer to constants declared in a surrounding block, function, method, or lambda body, if the initializer of the referenced constant declaration is itself a constant expression;
- Unary operators `'+'`, `'-'`, `'~'`, and `'!'`, but not `'++'` or `'--'` (see *Unary Plus*, *Unary Minus*, *Prefix Increment*, and *Prefix Decrement*);
- Casting conversions to numeric types (see *Cast Expression*);
- Multiplicative operators `'*'`, `'/'`, and `'%'` (see *Multiplicative Expressions*);
- Additive operators `'+'` and `'-'` (see *Additive Expressions*);
- Shift operators `'<<'`, `'>>'`, and `'>>>'` (see *Shift Expressions*);
- Relational operators `'<'`, `'<='`, `'>'`, and `'>='` (see *Relational Expressions*);
- Equality operators `'=='` and `'!='` (see *Equality Expressions*);
- Bitwise and logical operators `'&'`, `'^'`, and `'|'` (see *Bitwise and Logical Expressions*);
- Conditional-and operator `'&&'` (see *Conditional-And Expression*), and conditional-or operator `'||'` (see *Conditional-Or Expression*);
- Ternary conditional operator `condition?whenTrue:whenFalse` (see *Ternary Conditional Expressions*);
- Parenthesized expressions (see *Parenthesized Expression*) that contain constant expressions.

7.36.1 Specifics of Constant Expressions Evaluation

If a constant expression contains an unary or a binary operator applied to numeric operands, the operator is evaluated as follows:

- If any of operands are of type `double` or the operator is *exponentiation operator*, other operands are converted to type `double` before operator evaluation and the result type is `double`;
- If any of operands are of type `float`, other operands are converted to type `float` before operator evaluation and the result type is `float`; or
- Otherwise, all operands are converted to *some big integer type* that allows handling arbitrary-precision integers (like `bigint` type) or integers larger than the maximum value of type `long`.

If a constant expression consists of a single integer literal or a constant of an integer type, it is also converted to the same big integer type.

Type Inference for Constant Expressions is always used to infer the type of constant expression. In other words, *big integer type* is an internal compiler type and values of this type cannot occur during execution.

In case of mixed constant expression, each numeric subexpression is evaluated as described above:

```
const c = 3

c > 1 && c*2 < 8 // numeric subexpressions: (c > 1) and (c*2 < 8)
```

CHAPTER

EIGHT

STATEMENTS

Statements are designed to control execution.

Some statements are simple like a single *expressionStatement* (see *Expression Statements*), while others take several lines of program code like *Block* or *if Statements*.

```
...
i++    // Statement consists of single expression
...
if (i > 100) // 'if' statement starts
{          // 'block' statement starts, it is a part of 'if' statement
    i %= 100        // Statement belonging to 'block'
    console.log(i)  // Another statement belonging to 'block'
}   //  'block' statement ends, 'if' statement end
```

Note

The difference between statements and expressions is that *Expressions* evaluate a value of a certain type, while statements do not.

From the viewpoint of grammar rules, any expression ending in the semicolon ';' forms an *expressionStatement*. A missing semicolon is added automatically to an expression used as a statement.

A statement can consist of one or more expressions, or contain no expression at all. For example, the statement `i = 1` consists of an assignment expression that has the inferred type `int`, and the evaluated value `'1'`.

The syntax of *statements* is presented below:

```
statement:
    expressionStatement
    | block
    | constantOrVariableDeclaration
    | ifStatement
    | loopStatement
    | breakStatement
    | continueStatement
    | returnStatement
    | switchStatement
    | throwStatement
    | tryStatement
    ;
```

The list of statements in ArkTS is as follows:

Statement	Reference
Expression statements	*Expression Statements*
'{}' (block)	*Block*
let, *const* (variable or constant declarations)	*Constant Or Variable Declarations*
if-then-else	*if Statements*
while, *do*, *for*, *for-of*	*Loop Statements*
break	*break Statements*
continue	*continue Statements*
return	*return Statements*
switch	*switch Statements*
throw	*throw Statements*
try-catch-finally	*try Statements*

8.1 Normal and Abrupt Statement Execution

The actions that every statement performs in a normal mode of execution are specific for the particular kind of statement. Normal modes of evaluation for each kind of statement are described in the following sections.

A statement execution is considered to *complete normally* if the desired action is performed without an error being thrown. On the contrary, a statement execution is considered to *complete abruptly* if it causes an error thrown.

8.2 Expression Statements

Any expression can be used as a statement.

The syntax of an *expression statement* is presented below:

```
expressionStatement:
    expression
    ;
```

The execution of a statement leads to the execution of the expression. The result of such execution is discarded.

8.3 Block

A sequence of statements (see *Statements*) enclosed in balanced braces forms a *block*.

The syntax of a *block statement* is presented below:

```
block:
    '{' statement* '}'
    ;
```

The execution of a block means that all block statements, except type declarations, are executed one after another in the textual order of their appearance within the block while an error is thrown (see *Errors*), or until a return occurs (see *return Statements*).

If a block is the body of a `functionDeclaration` (see *Function Declarations*) or a `classMethodDeclaration` (see *Method Declarations*) declared implicitly or explicitly with return type `void` (see *Type void or undefined*), then the block can contain no return statement at all. Such a block is equivalent to one that ends in a `return` statement, and is executed accordingly.

8.4 Constant Or Variable Declarations

Constant or variable declarations define new mutable or immutable variables within the enclosing context.

`Let` and `const` declarations have the initialization part that presumes execution, and actually act as statements.

The syntax of a *constant or variable declaration* is presented below:

```
constantOrVariableDeclaration:
    annotationUsage?
    ( variableDeclaration
    | constantDeclaration
    )
    ;
```

The visibility of declaration name is determined by the surrounding module, namespace, function, or method, and by the block scope rules (see *Scopes*). In order to avoid ambiguous interpretation, appropriate sections of this Specification are dedicated to a detailed discussion of the following entities:

- *if Statements*,
- *for Statements*,
- *for-of Statements*.

Any declaration can shadow another same-name declaration (if any) in the same surrounding scope.

```
let item: number = 123
{
   const item: string = "123"
   // This 'item' is of type 'string'
}
// This 'item' is of type 'number'

```

(continues on next page)

(continued from previous page)

```
function foo (item: boolean) {
   // this 'item' is of type 'number'
   let item: number[] = [] // Compile-time error as parameter 'item' and
                           // local variable 'item' lead to duplication as
                           // they are in the same scope
}
```

The usage of annotations is discussed in *Using Annotations*.

8.5 `if` Statements

An `if` statement allows executing alternative statements (if provided) under certain conditions.

The syntax of an *if statement* is presented below:

```
ifStatement:
    'if' '(' expression ')' thenStatement
    ('else' elseStatement)?
    ;

thenStatement:
    statement
    ;

elseStatement:
    statement
    ;
```

Type of expression must be `boolean`, or a type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

If an expression is successfully evaluated as `true`, then a `thenStatement` is executed. Otherwise, an `elseStatement` is executed (if provided).

Any `else` corresponds to the nearest preceding `if` of an `if` statement:

```
if (Cond1)
if (Cond2) statement1
else statement2 // Executes only if: Cond1 && !Cond2
```

A *Block* can be used to combine the `else` part with the initial `if` as follows:

```
if (Cond1) {
  if (Cond2) statement1
}
else statement2 // Executes if: !Cond1
```

If `thenStatement` or `elseStatement` is any kind of a statement but not a block (see *Block*), then no *block scope* (see *Scopes*) is created for such a statement.

```
function foo(Cond1: boolean) {
  if (Cond1) let x: number = 1
  x = 2 // OK

  if (Cond1) {
    let x: number = 10;   // OK, then-block scope
    let y: number = x;
  }
  else {
    let x: number = 20   // OK, no conflict, else-block scope
    y = x;           // Compile-time error, no 'y' in scope
  }

  console.log(x)  // OK, prints 2
  console.log(y)  // Compile-time error, 'y' unknown
}
```

8.6 Loop Statements

ArkTS has four kinds of loops. A loop of each kind can be optionally labeled with an *identifier*. The *identifier* can be used only by the *break Statements* and *continue Statements* contained in the loop body.

The syntax of *loop statements* is presented below:

```
loopStatement:
    (identifier ':')?
    whileStatement
    | doStatement
    | forStatement
    | forOfStatement
    ;
```

A compile-time error occurs if the label *identifier* is not used within `loopStatement`, or is used in lambda expressions (see *Lambda Expressions*) within a loop body.

```
label: for (i = 1; i < 10; i++) {
    const f1 = () => {
        while (true) {
            continue label // Compile-time error
        }
    }
    const f2 = () => {
        do
            break label // Compile-time error
        while (true)
    }
}
```

8.7 while Statements and do Statements

A while statement and a do statement evaluate an expression and execute the statement repeatedly till the expression value is true. The key difference is that a whileStatement starts from evaluating and checking the expression value, and a doStatement starts from executing the statement.

The syntax of *while and do statements* is presented below:

```
whileStatement:
    'while' '(' expression ')' statement
    ;

doStatement
    : 'do' statement 'while' '(' expression ')'
    ;
```

Type of expression must be boolean, or a type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

8.8 for Statements

The syntax of *for statements* is presented below:

```
forStatement:
    'for' '(' forInit? ';' forContinue? ';' forUpdate? ')' statement
    ;

forInit:
    expressionSequence
    | variableDeclarations
    ;

forContinue:
    expression
    ;

forUpdate:
    expressionSequence
    ;
```

Type of *forContinue* expression must be boolean, or a type mentioned in *Extended Conditional Expressions*. Otherwise, a compile-time error occurs.

```
// The existing variable is used as a loop index variable
let i: number
for (i = 1; i < 10; i++) {
```

(continues on next page)

(continued from previous page)

```
  console.log(i)
}

// A new variable is declared as a loop index variable with its type
// explicitly specified
for (let i: int = 1; i < 10; i++) {
  console.log(i)
}

// A new variable is declared as a loop index variable with its type
// inferred from its initialization part of the declaration
for (let i = 1; i < 10; i++) {
  console.log(i)
}
```

A variable declared in the *forInit*-part has the loop scope. It can be used in a `forContinue` expression, a `forUpdate` expression, a single-body statement, or in a body block:

```
// forInit declaration and no body block
let k: int = 0
for (let i: int = 1; i < 10; i++)
  k += i
console.log(k)
// i =  k  // Compile-time error if uncommented
let i: int = k  // OK
```

8.9 `for-of` Statements

A `for-of` loop iterates elements of an instance of an *iterable type* (see *Iterable Types*) and executes the loop body having these elements available.

The syntax of *for-of statements* is presented below:

```
forOfStatement:
    'for' '(' forVariable 'of' expression ')' statement
    ;

forVariable:
    identifier | ('let' | 'const') identifier (':' type)?
    ;
```

If type of an expression is not iterable, then a compile-time error occurs.

The execution of a `for-of` loop starts from the evaluation of `expression`. Then, if the evaluation is successful, for every loop iteration *forVariable* is set to the successive element as a result of iterator advancement, and the loop body (i.e., `statement`) is executed.

If *forVariable* contains modifiers `let` or `const`, then *forVariable* declares a new variable accessible inside the loop body only. Otherwise, the variable declared elsewhere is used.

The modifier `const` forbids assignment into *forVariable*, while the modifier `let` allows modifications.

The type of *forVariable* declared inside the loop is inferred to be that of the *iterated* elements, namely:

- `T`, if `Array<T>` or `FixedArray<T>` instance is iterated;
- `string`, if a `string` value is iterated;
- Type argument of the *iterator*, if an instance of the *iterable* type is iterated.

If *forVariable* is declared outside the loop, then the type of an iterated element must be assignable (see *Assignability*) to the type of the variable. Otherwise, a compile-time error occurs.

```
// The existing variable 's'
let s : string
for (s of "a string object") {
  console.log(s)
}

// New variable 's', its type is inferred from the expression after 'of'
for (let s of "a string object") {
  console.log(s)
}

// New variable 'element', its type is inferred from the expression after 'of'
// as 'const' it cannot be assigned a new value in the loop body
for (const element of [1, 2, 3]) {
  console.log(element)
  element = 66 // Compile-time error as 'element' is 'const'
}
```

Explicit type annotation of *forVariable* is allowed as an experimental feature (see *For-of Explicit Type Annotation*).

8.10 `break` Statements

A `break` statement transfers control out of the enclosing `loopStatement` or `switchStatement`. If a `break` statement is used outside a `loopStatement` or a `switchStatement`, then a compile-time error occurs.

The syntax of a *break statement* is presented below:

```
breakStatement:
    'break' identifier?
    ;
```

A `break` statement with the label *identifier* transfers control out of the enclosing statement with the same label *identifier*. If there is no enclosing loop statement with the same label identifier (within the body of the surrounding function or method), then a compile-time error occurs.

A statement without a label transfers control out of the innermost enclosing `switch`, `while`, `do`, `for`, or `for-of` statement. If `breakStatement` is placed outside `loopStatement` or `switchStatement`, then a compile-time error occurs.

Examples of `break` statements with and without a label are presented below:

```
// A single iteration
while (true) {
  console.log("iteration")  // Gets printed exactly once
  break;
}

let a: number = 0
outer:
  do {
    for (a = 0; a < 10; a++) {
        if (a == 1) break outer
        console.log("inner")    // Gets printed once only
    }
    console.log(a) // Never reached
  } while (true)   // Condition never used
```

8.11 continue Statements

A continue statement terminates the execution of a current loop iteration, and transfers control to the next iteration while keeping the appropriate checks of the loop exit conditions in place.

The syntax of a *continue statement* is presented below:

```
continueStatement:
    'continue' identifier?
    ;
```

A continue statement with no label transfers control to the next iteration of the enclosing loop statement. If there is no enclosing loop statement within the body of the surrounding function or method, then a compile-time error occurs.

A continue statement with the label *identifier* transfers control to the next iteration of the enclosing loop statement with the same label *identifier*. If there is no enclosing loop statement with the same label *identifier* (within the body of the surrounding function or method), then a compile-time error occurs.

Examples of continue statements with and without a label are presented below:

```
// continue     // Compile-time error if uncommented

// Continue without label
// Prints 0, 1, 2, 4 (3 skipped)
for (let a: int = 0; a < 5; a++){
  if (a == 3) continue
  console.log("a = " + a)
}

let a: number
outer:
  do {
    for (a = 0; a < 10; a++) {
        if (a > 1) continue outer
```

(continues on next page)

(continued from previous page)

```
        console.log("inner")    // Gets printed twice only
    }
    console.log("Outer") // Never reached
  } while (false)
```

8.12 `return` Statements

A `return` statement terminates the execution of a current function, method, constructor, or lambda, and returns control to the caller. In case of a function, method, and lambda call execution of a `return` statement implies that the call returns a value defined as a result of the *return expression* evaluation.

The syntax of a *return statement* is presented below:

```
returnStatement:
    'return' expression?
    ;
```

If no *expression* is provided, then a *return statement* is semantically equivalent to the form `return undefined`.

A *return statement* in the plain form `return` (with no *expression*) can occur inside one of the following:

- Constructor body;
- Function, method, or lambda body with return type `void` or `undefined` (see *Type void or undefined*), or a union type (see *Union Types*) containing `void` or `undefined`;
- Asynchronous function, method or lambda body with return type `Promise<void>` (see *Asynchronous execution*);

Otherwise, a compile-time error occurs.

A compile-time error also occurs if type of a return expression is not assignable (see *Assignability*) to the surrounding function, method, or lambda return type.

The semantics is represented in the examples below:

```
return // Compile-time error, top-level statements cannot contain a return statement
namespace NS {
   return // Compile-time error, top-level statements cannot contain a return statement
}
function f1 () {} // OK, no return statement equals the form 'return undefined'
function f2 () { return } // OK, 'return' equals the form 'return undefined'
function f3 (): void { return undefined } // Full syntax form

function f4(cond: boolean): Base {
    if (cond) {
        return new Derived // OK, as Derived is assignable to Base
    } else {
        return new Object // Compile-time error, as Object is not assignable to Base
    }
}
```

(continues on next page)

(continued from previous page)

```
class A {
    constructor () {
        return  // OK
    }
    constructor (p: number) {
        return undefined  // Compile-time error, such syntax form is forbidden
    }
}
```

8.13 `switch` Statements

A `switch` statement transfers control to a statement or a block by using the result of successful evaluation of the value of a `switch` expression.

The syntax of a *switch statement* is presented below:

```
switchStatement:
    (identifier ':')? 'switch' '(' expression ')' switchBlock
    ;

switchBlock
    : '{' caseClause* defaultClause? caseClause* '}'
    ;

caseClause
    : 'case' expression ':' statement*
    ;

defaultClause
    : 'default' ':' statement*
    ;
```

A `switch` expression can be of any type.

If available, an optional identifier allows the `break` statement to transfer control out of a nested `switch` or `loop` statement (see *break Statements*).

A compile-time error occurs if at least one of case expression types is not assignable (see *Assignability*) to the type of the `switch` statement expression.

```
let arg: string = prompt("Enter a value?");
switch (arg) {
  case '0':
  case '1':
    console.log('One or zero')
    break
  case '2':
    console.log('Two')
```

(continues on next page)

(continued from previous page)

```
    break
  default:
    console.log('An unknown value')
}

class A {}
let a: A| null = assignIt()
switch (a) {
  case null:
  case null: // One may have several case clauses with the same expression in
    console.log ("a is null")
    break
  case new A:
    console.log ("Never matches as new A is a new unique object")
    break
  default:
    console.log ("a is A")
}
function assignIt () {
    return new A
}
```

The execution of a `switch` statement starts from the evaluation of the `switch` expression.

The value of the `switch` expression is compared repeatedly to the value of case expressions. The comparison starts from the top and proceeds until the first *match*. A *match* occurs when a particular case expression value equals the value of the `switch` expression in terms of the operator `'=='`. The execution is transferred to the set of statements of the *caseClause* where the match occurred. If this set of statements executes a `break` statement, then the entire `switch` statement terminates. If no `break` statement is executed, then the execution continues through statements of any remaining *caseClause* and *defaultClause* until the first `break` statement occurs, or until the `switch` statement ends.

If no *match* occurs while a *defaultClause* is present, then the execution is transferred to the statements of the *defaultClause*.

8.14 `throw` Statements

A `throw` statement causes an *error* object to be created and raised (see *Error Handling*). It immediately transfers control, and can exit multiple statements, constructors, functions, and method calls until a `try` statement (see *try Statements*) is found that catches the value thrown. If no `try` statement is found, then `UncaughtExceptionError` is thrown.

The syntax of a *throw statement* is presented below:

```
throwStatement:
    'throw' expression
    ;
```

The expression type must be assignable (see *Assignability*) to type `Error`. Otherwise, a compile-time error occurs.

This implies that `null` or `undefined` cannot be thrown.

Errors can be thrown at any place in the code.

8.15 `try` Statements

A `try` statement runs block of code, and provides optional `catch` clause to handle errors (see *Error Handling*) which may occur during block of code execution.

The syntax of a *try statement* is presented below:

```
tryStatement:
      'try' block catchClause? finallyClause?
      ;

catchClause:
      'catch' '(' identifier ')' block
      ;

finallyClause:
      'finally' block
      ;
```

A `try` statement must contain:

- `finally` clause;
- `catch` clause, or
- Both a `finally` clause and a `catch` clause.

Otherwise, a compile-time error occurs.

If the `try` block completes normally, then no `catch` clause block is executed, if present.

If an error is thrown in the `try` block directly or indirectly, then the control is transferred to the `catch` clause, if present.

8.15.1 `catch` Clause

A `catch` clause consists of two parts:

- A *catch identifier* that provides access to an object associated with the *error* thrown; and
- A block of code that handles the error.

The type of *catch identifier* inside the block is `Error` (see *Error Handling*).

```
class ZeroDivisor extends Error {}

function divide(a: number, b: number): number {
  if (b == 0)
    throw new ZeroDivisor()
```

(continues on next page)

(continued from previous page)

```
  return a / b
}

function process(a: number, b: number): number {
  try {
    let res = divide(a, b)

    // Further processing ...
    return res
  }
  catch (e) {
    return e instanceof ZeroDivisor? -1 : 0
  }
}
```

A `catch` clause handles all errors at runtime. It returns '*-1*' for the `ZeroDivisor`, and '*0*' for all other errors.

8.15.2 `finally` Clause

A `finally` clause defines the set of actions in the form of a block to be executed without regard to whether a `try-catch` completes normally or abruptly.

The syntax of a *finally clause* is presented below:

```
finallyClause:
    'finally' block
    ;
```

A `finally` block is executed without regard to how (by reaching `return` or `try-catch` end or raising new *error*) the program control is transferred out. The `finally` block is particularly useful to ensure proper resource management.

Any required actions (e.g., flush buffers and close file descriptors) can be performed while leaving the `try-catch`:

```
class SomeResource {
  // Some class members
  // ...
  close() {}
}

function ProcessFile(name: string) {
  let r = new SomeResource()
  try {
    // Some processing
  }
  finally {
    // Finally clause is executed after try-catch
        completes normally or abruptly
    r.close()
  }
}
```

8.15.3 `try` Statement Execution

1. A `try` block and the entire `try` statement complete normally if no `catch` block is executed. The execution of a `try` block completes abruptly if an error is thrown inside the `try` block.
2. The execution of a `try` block completes abruptly if error *x* is thrown inside the `try` block. If the `catch` clause is present, and the execution of the body of the `catch` clause completes normally, then the entire `try` statement completes normally. Otherwise, the `try` statement completes abruptly.
3. If no `catch` clause is in place, then the error is propagated to the surrounding and caller scopes until reaching the scope with the `catch` clause to handle the error. Subsequent steps are then defined by the execution environment.
4. If `finally` clause is in place, and its execution completes abruptly, then the `try` statement also completes abruptly.

CHAPTER

NINE

CLASSES

Class declarations introduce new reference types and describe the manner of their implementation.

A class body contains declarations and initializer blocks.

Declarations can introduce class members (see *Class Members*) or class constructors (see *Constructor Declaration*).

The body of the declaration of a member comprises the scope of a declaration (see *Scopes*).

Class members include:

- Fields,
- Methods, and
- Accessors.

Class members can be *declared* or *inherited*.

Every member is associated with the class declaration it is declared in.

Field, method, accessor and constructor declarations can have the following access modifiers (see *Access Modifiers*):

- `Public`,
- `Protected`,
- `Private`.

Every class defines two class-level scopes (see *Scopes*): one for instance members, and the other for static members. It means that two members of a class can have the same name if one is static while the other is not.

9.1 Class Declarations

Every class declaration defines a *class type*, i.e., a new named reference type.

The class name is specified by an *identifier* inside a class declaration.

If `typeParameters` are defined in a class declaration, then that class is a *generic class* (see *Generics*).

The syntax of *class declaration* is presented below:

```
classDeclaration:
    classModifier? 'class' identifier typeParameters?
    classExtendsClause? implementsClause?
```

(continues on next page)

(continued from previous page)

```
    classMembers
    ;

classModifier:
    'abstract' | 'final'
    ;
```

Classes with the `final` modifier are an experimental feature discussed in *Final Classes*.

The scope of a class declaration is specified in *Scopes*.

An example of a class is presented below:

```
class Point {
  public x: number
  public y: number
  public constructor(x : number, y : number) {
    this.x = x
    this.y = y
  }
  public distanceBetween(other: Point): number {
    return Math.sqrt(
      (this.x - other.x) * (this.x - other.x) +
      (this.y - other.y) * (this.y - other.y)
    )
  }
  static origin = new Point(0, 0)
}
```

9.1.1 Abstract Classes

A class with the modifier `abstract` is known as abstract class. An abstract class is a class that cannot be instantiated, i.e., no objects of this type can be created. It serves as a blueprint for other classes by defining common fields and methods that subclasses must implement. Abstract classes can contain both abstract and concrete methods.

A compile-time error occurs if an attempt is made to create an instance of an abstract class:

```
abstract class X {
   field: number
   constructor (p: number) { this.field = p }
}
let x = new X(42)
  // Compile-time error, Cannot create an instance of an abstract class.
```

Subclasses of an abstract class can be abstract or non-abstract. A non-abstract subclass of an abstract superclass can be instantiated. As a result, a constructor for the abstract class, and field initializers for non-static fields of that class are executed:

```
abstract class Base {
   field: number
```

(continues on next page)

(continued from previous page)

```
   constructor (p: number) { this.field = p }
}

class Derived extends Base {
   constructor (p: number) { super(p) }
}
```

A method with the modifier `abstract` is considered an *abstract method* (see *Abstract Methods*). Abstract methods have no bodies, i.e., they can be declared but not implemented.

Only abstract classes can have abstract methods. A compile-time error occurs if a non-abstract class has an abstract method:

```
class Y {
  abstract method (p: string)
  /* compile-time error, Abstract methods can only
     be within an abstract class. */
}
```

A compile-time error occurs if an abstract method declaration contains the modifiers `final` or `override`.

```
abstract class Y {
  final abstract method (p: string)
  // Compile-time error, Abstract methods cannot be final
}
```

9.2 Class Extension Clause

All classes except class `Object` can contain the `extends` clause that specifies the *base class*, or the *direct superclass* of the current class. In this situation, the current class is a *derived class*, or a *direct subclass*. Any class, except class `Object` that has no `extends` clause, is assumed to have the `extends Object` clause.

The syntax of *class extension clause* is presented below:

```
classExtendsClause:
    'extends' typeReference
    ;
```

A compile-time error occurs if:

- `typeReference` refers directly to, or is an alias of any non-class type, e.g., of interface, enumeration, union, function, utility type, or any instantiation of `FixedArray`.
- `typeReference` names a class type that is not accessible (see *Accessible*).
- `extends` clause appears in the declaration of the class `Object`.
- `extends` graph has a cycle.

Class extension implies that a class inherits all members of the direct superclass.

Note

Private members are inherited from superclasses, but are not accessible (see *Accessible*) within subclasses:

```
class Base {
  /* All methods are accessible in the class where
      they were declared */
  public publicMethod () {
    this.protectedMethod()
    this.privateMethod()
  }
  protected protectedMethod () {
    this.publicMethod()
    this.privateMethod()
  }
  private privateMethod () {
    this.publicMethod();
    this.protectedMethod()
  }
}
class Derived extends Base {
  foo () {
    this.publicMethod()    // OK
    this.protectedMethod() // OK
    this.privateMethod()   // Compile-time error,
                           // the private method is inaccessible
  }
}
```

The transitive closure of a *direct subclass* relationship is the *subclass* relationship. Class `A` can be a subclass of class `C` if:

- Class `A` is the direct subclass of `C`; or
- Class `A` is a subclass of some class `B`, which is in turn a subclass of `C` (i.e., the definition applies recursively).

Class `C` is a *superclass* of class `A` if `A` is its subclass.

9.3 Class Implementation Clause

A class can implement one or more interfaces. Interfaces to be implemented by a class are listed in the `implements` clause. Interfaces listed in this clause are *direct superinterfaces* of the class.

The syntax of *class implementation clause* is presented below:

```
implementsClause:
    'implements' interfaceTypeList
    ;

interfaceTypeList:
```

(continues on next page)

(continued from previous page)

```
    typeReference (',' typeReference)*
    ;
```

If `typeReference` fails to name an accessible interface type (see *Accessible*), then a compile-time error occurs.

```
// File1
interface I { } // Not exported

// File2
class C implements I {} // Compile-time error I is not accessible
```

If some interface is repeated as a direct superinterface in a single `implements` clause (even if that interface is named differently), then all repetitions are ignored.

For the class declaration `C<`F_1`, ..., `F_n`>` ($n \geq 0, C \neq Object$):

- *Direct superinterfaces* of class type `C<`F_1`, ..., `F_n`>` are the types specified in the `implements` clause of the declaration of `C` (if there is an `implements` clause).

For the generic class declaration `C<`F_1`, ..., `F_n`>` ($n > 0$):

- *Direct superinterfaces* of the parameterized class type `C<`T_1`, ..., `T_n`>` are all types `I<`$U_1\theta$`, ..., `$U_k\theta$`>` if:
  - T_i ($1 \leq i \leq n$) is a type;
  - `I<`U_1`, ..., `U_k`>` is the direct superinterface of `C<`F_1`, ..., `F_n`>`; and
  - θ is the substitution [F_1 `:=` T_1 `, ...,` F_n `:=` T_n].

Interface type `I` is a superinterface of class type `C` if `I` is one of the following:

- Direct superinterface of `C`;
- Superinterface of `J` which is in turn a direct superinterface of `C` (see *Superinterfaces and Subinterfaces* that defines superinterface of an interface); or
- Superinterface of the direct superclass of `C`.

A class *implements* all its superinterfaces.

A compile-time error occurs if:

- Class implements two interface types that represent different instantiations of the same generic interface (see *Generics*);
- Class field and a method inherited from one of its superinterfaces have the same name, except where either is static and the other is not.

If a class is not *abstract*, then the following conditions must be met:

- Methods of a superinterface have implementation bodies that can be defined either in the class itself, or in its superclass or superinterface (see *Overriding in Classes* for detail);
- Required properties of superinterfaces are implemented (see *Implementing Required Interface Properties*).
- Otherwise, a compile-time error occurs.

An optional property from a superinterface can be implemented or implicitly defined (see *Implementing Optional Interface Properties*).

9.3.1 Implementing Required Interface Properties

A class must implement all required properties from all superinterfaces (see *Interface Properties*) that can be defined as one of the following:

- Field,
- Readonly field,
- Getter,
- Setter, or
- Both a getter and a setter.

The following table summarizes all valid variants of implementation, and a compile-time error occurs for any other combinations:

Form of Interface Property	Implementation in a Class
field	field, or getter and setter
readonly field	readonly field, field, getter, or getter and setter
getter only	readonly field, field, getter, or getter and setter
setter only	field, setter, or setter and getter
getter and setter	field, or getter and setter

If a class field is used to implement an interface property in any form, then:

- The field is represented as an instance data member (see *Field Declarations*);
- Accessing the field via a class instance always means accessing an ordinary class field;
- Accessing a property via a reference of an interface type always means calling a getter or a setter;
- If an interface property is defined in a field form, then a getter and a setter (if the property is not `readonly`) are generated implicitly in the class by the compiler.

This is represented in the following example:

```
interface I {
    n: number
}
class C implements I {
    n: number = 1 // property is implemented as a field
}
let c = new C()
console.log(c.n) // class field read
c.n = 1 // class field write

let i: I = c
console.log(c.n) // implicitly generated getter is called
i.n = 2          // implicitly generated setter is called
```

Getter and setter generated implicitly look as follows:

```
get n(): number  { return this.n }
set n(x: number) { this.n = x }
```

Note

- `this.n` in the pseudocode above is generated to provide access to the class field but not as a call to a getter or a setter;
- Attempting to explicitly declare a getter or a setter with the same name as a field causes a compile-time error because class members must be *distinguishable* (see *Declarations*).

An interface property implemented as an accessor or accessors is represented in the example below:

```
interface I {
    n: number
    readonly r: string
}
class C implements I {
    // 'n' is explicitly implemented as getter and setter
    get n(): number  { return 1 }
    set n(x: number) { /*some body*/ }
    // 'r' is explicitly implemented as getter
    get r(): string { return "abc" }
    // a setter can be defined for 'r', but it is not mandatory
    set r(x: string) { /*some body*/ }
}
```

A compile-time error occurs if:

- Interface property is `readonly`, and the getter is not defined;
- Interface property is not `readonly`, and either a getter or a setter is not defined.

The errors are represented in the example below:

```
interface I {
    n: number
}
class A implements I { // Compile-time error, setter is not defined
    get n(): number  { return 1 }
}
class B implements I { // Compile-time error, getter is not defined
    set n(x: number)  {}
}
class C implements I {
    get n(): string { return "aa" } // Compile-time error, wrong type
}

interface J {
    readonly r: string
}
class D implements J { // Compile-time error, getter is not defined
    set r(x: string) {}
}
```

If an interface property is defined in the form of an accessor (a getter, a setter, or both) and implemented by a class field, then implicit accessor bodies are generated in the class but only for accessors defined in the interface. The situation is represented below:

```
interface I {
  get n(): number
}
class C implements I {
    n: number = 1 // property is implemented as a field
    // getter body is implicitly generated
    // setter is not defined and not generated
}

function foo(i: I) {
  console.log(i.n) // OK, getter is used
  i.n = 1 // Compile-time error, setter is not defined
}

interface J {
  get n(): number
  set n(x: number)
}
class D implements J {
    n: number = 1 // property is implemented as a field
    // getter body is implicitly generated
    // setter body is implicitly generated
}

function bar(j: J) {
  console.log(j.n) // OK, getter is used
  j.n = 1          // OK, setter is used
}
```

A compile-time error occurs if **all** of the following conditions are fulfilled:

- Interface defines both a getter and a setter that have the same name;
- Class implements a property by a field;
- Return type of a getter and parameter type of a setter are not of the same type.

This situation is represented by the example below:

```
interface J {
  get n(): number
  set n(x: string)
}
class D implements J {
    n: number = 1 // Compile-time error, types mismatch
}
```

If a property defines both a getter and a setter of different types, then the property can be implemented by accessors:

```
interface J {
  get n(): number
  set n(x: string)
}

class E implements J {
```

(continues on next page)

(continued from previous page)

```
  value: string = ""
  get n(): number  { return Number.parseFloat(this.value) }
  set n(x: string) { this.value = x }
}

let e = new E
e.n = "123"
console.log(e.n)
```

If a superclass implements an interface property, then all derived classes inherit the property in the same form. A compile-time error occurs on an attempt to do one of the following:

- Overriding a superclass field by an accessor or accessors;
- Overriding a superclass accessor by a field.

Such error situations are represented by the example below:

```
interface I {
    n: number
}
class C implements I {
    n: number = 1
}
class C1 extends C {
    get n(): number { return 1 } // Compile-time error, 'n' cannot be overridden by an accessor
}

class D implements I {
    get n(): number { return 1 }
    set n(x: number) {}
}
class D1 extends D {
    n: number = 2 // Compile-time error, 'n' cannot be overridden by a field
}
```

A field that implements an interface property can override a field of the same type defined in a superclass:

```
interface I {
    n: number
}
class A {
    n: number = 1
}
class C extends A implements I {
    n: number = 2 // OK, 'n' overrides 'n' from A and implements 'n' from I
}
```

If types mismatch as represented in the example below, then a compile-time error occurs:

```
interface I {
    n: number
}
```

(continues on next page)

(continued from previous page)

```
interface J {
    n: string
}
class A implements I, J { // Compile-time error, types mismatch
}

class B {
    n: string = "aa"
}
class C extends B implements I { // Compile-time error, types mismatch
}

class C implements I {
    get n(): string { return "aa" } // Compile-time error, types mismatch
}
```

If a property is defined as `readonly`, then the implementation of the property can be `readonly` or not `readonly` as follows:

```
interface I {
    readonly r: number
}
class A implements I {
    r: number = 0 // OK, the field is writeable
}
class B implements I {
    readonly r: number = 1  // OK, the field is readonly
}

function foo(i: I) {
    i.r = 42 // Compile-time error, 'r' is readonly
    if (i instanceof A) {
        i.r = 42 // OK, here 'i' is of type A, and 'r' is writeable
    }
    if (i instanceof B) {
     i.r = 42 // Compile-time error, here 'i' is of type B, and 'r' is readonly
    }
}
```

If a writeable interface property is implemented as `readonly` as represented in the example below, then a compile-time error occurs:

```
interface I {
    r: number
}
class C implements I {
    readonly r: number = 1 // Compile-time error
}

interface J {
    readonly r: number
}
```

(continues on next page)

(continued from previous page)

```
class D implements I, J {
    readonly r: number = 1 // Compile-time error
}
class E implements I, J {
    r: number = 1 // OK
}
```

If no implementation for the property is provided, then a compile-time error occurs:

```
interface I {
    property: number
}
class C implements I { // Compile-time error, no implementation at all
}
```

9.3.2 Implementing Optional Interface Properties

A class can implement *Optional Interface Properties*) from superinterfaces or use implicitly defined accessors from an interface.

The use of accessors implicitly defined in the interface is represented in the example below:

```
interface I {
  n?: number
}
class C implements I {}

let c = new C()
console.log(c.n) // Output: undefined
c.n = 1 // runtime error is thrown
```

The implementation of an optional interface property as a field is represented in the example below:

```
interface I {
  num?: number
}
class C implements I {
  num?: number = 42
}
```

For the example above, the private hidden field and the required accessors are defined implicitly for the class C overriding accessors from the interface:

```
class C implements I {
  private $$_num: number = 42 // the exact name of the field is implementation specific
  get num(): number | undefined { return this.$$_num }
  set num(n: number | undefined) { this.$$_num = n }
}
```

If a property is implemented by accessors (see *Class Accessor Declarations*), then it is acceptable to implement only one accessor for an optional field, and use default implementation for another accessor as represented in the following example:

```
interface I {
  num?: number
}

class C1 implements I { // OK, both default implementations
}

class C2 implements I { // OK, default implementation used for get
  set num(n: number | undefined) { this.$$_num = n }
}

class C3 implements I { // OK, both explicit implementations
  get num(): number | undefined { return this.$$_num }
  set num(n: number | undefined) { this.$$_num = n }
}
```

A compile-time error occurs, if an optional property in an interface is implemented as non-optional field:

```
interface I {
  num?: number
}
class C implements I {
  num: number = 42 // Compile-time error, must be optional
}
```

9.4 Class Members

A class can contain declarations of the following members:

- Fields,
- Methods,
- Accessors,
- Constructors,
- Method overloads (see *Explicit Class Method Overload*), and
- Single static initialization block (see *Static Initialization*).

The syntax is presented below:

```
classMembers:
    '{'
       classMember* staticBlock? classMember*
    '}'
    ;
```

(continues on next page)

(continued from previous page)

```
classMember:
    annotationUsage?
    accessModifier?
    ( constructorDeclaration
    | explicitConstructorOverload
    | classFieldDeclaration
    | classMethodDeclaration
    | explicitClassMethodOverload
    | classAccessorDeclaration
    )
    ;

staticBlock: 'static' Block;
```

Declarations can be inherited or immediately declared in a class. Any declaration within a class has a class scope. The class scope is fully defined in *Scopes*.

Members can be static or non-static as follows:

- Static members that are not part of class instances, and can be accessed by using a qualified name notation (see *Names*) anywhere the class name is accessible (see *Accessible*); and
- Non-static, or instance members that belong to any instance of the class.

Note

Static members of a superclass are not inherited. A subclass that is to use static members of its superclass must use explicit qualification by the the name of the superclass. It applies to any kind of static members:

- static fields;
- static methods;
- static accessors.

Names of all *accessible* static and non-static entities in a class declaration scope (see *Scopes*) must be unique, i.e., fields, methods, and overloads with the same static or non-static status cannot have the same name.

The use of annotations is discussed in *Using Annotations*.

Class members are as follows:

- Members inherited from their direct superclass (see *Inheritance*), except class `Object` that cannot have a direct superclass.
- Members declared in a direct superinterface (see *Superinterfaces and Subinterfaces*).
- Members declared in the class body (see *Class Members*).

Class members declared `private` are not accessible (see *Accessible*) to all subclasses of the current class.

Class members declared `protected` or `public` are inherited by all subclasses of the class and accessible (see *Accessible*) for all subclasses.

Constructors and static block are not members, and are not inherited.

Members can be as follows:

- Class fields (see *Field Declarations*),

- Methods (see *Method Declarations*), and
- Accessors (see *Class Accessor Declarations*).

9.5 Access Modifiers

Access modifiers define how a class member or a constructor can be accessed. Accessibility in ArkTS can be of the following kinds:

- `Private`,
- `Protected`,
- `Public`.

The desired accessibility of class members and constructors can be explicitly specified by the corresponding *access modifiers*.

The syntax of *class members or constructors modifiers* is presented below:

```
accessModifier:
    'private'
    | 'protected'
    | 'public'
    ;
```

If no explicit modifier is provided, then a class member or a constructor is implicitly considered `public` by default.

9.5.1 Private Access Modifier

The modifier `private` indicates that a class member or a constructor is accessible (see *Accessible*) within its declaring class only, i.e., a private member or constructor *m* declared in some class `C` can be accessed only within the class body of `C`. The usage of `private` is represented in the following example:

```
class C {
  private count: number = 1
  private hello() {}
  getCount(): number {
    return this.count // OK
  }
  sayHello() {
    this.hello()  // OK
  }
}

function increment(c: C) {
  c.sayHello()  // OK
  c.hello()     // Compile-time error - 'hello()' is private
```

(continues on next page)

(continued from previous page)

```
  c.count++     // Compile-time error - 'count' is private
}

class D1 extends C {
  foo() {
     this.getCount() // OK
     this.sayHello() // OK
     this.hello()    // Compile-time error, private base method
     this.count++    // Compile-time error, private base field
  }
}

class D2 extends C {
  count = 1  // OK to reuse name since `count` in base inaccessible
  hello() {} // OK to reuse name since `hello` in base inaccessible
  foo() {
     this.count++    // OK
     this.hello()    // OK
  }
}

function bar(p: D2) {
  p.count++ // OK
  p.hello() // OK
}
```

9.5.2 Protected Access Modifier

The modifier `protected` indicates that a class member or a constructor is accessible (see *Accessible*) only within its declaring class and the classes derived from that declaring class. A protected member `M` declared in some class `C` can be accessed only within the class body of `C` or of a class derived from `C`:

```
class C {
  protected count: number
   getCount(): number {
     return this.count // OK
   }
}

class D extends C {
  increment() {
    this.count++ // OK, D is derived from C
  }
}

function increment(c: C) {
  c.count++ // Compile-time error - 'count' is not accessible
}
```

9.5.3 Public Access Modifier

The modifier `public` indicates that a class member or a constructor can be accessed everywhere, provided that the member or the constructor belongs to a type that is also accessible (see *Accessible*).

9.6 Field Declarations

Field declarations represent data members in class instances or static data members (see *Static and Instance Fields*). Class instance *field declarations* are its *own fields* in contrast to the inherited ones. Syntactically, a field declaration is similar to a variable declaration.

```
classFieldDeclaration:
    fieldModifier*
    identifier
    ( '?'? ':' type initializer?
    | '?'? initializer
    | '!' ':' type
    )
    ;

fieldModifier:
    'static' | 'readonly' | 'override'
    ;
```

A field with an identifier marked with `'?'` is called *optional field* (see *Optional Fields*). A field with an identifier marked with `'!'` is called *field with late initialization* (see *Fields with Late Initialization*).

A compile-time error occurs if:

- Some field modifier is used more than once in a field declaration.
- Name of a field declared in the body of a class declaration is also used for a method of this class with the same static or non-static status.
- Name of a field declared in the body of a class declaration is also used for another field in the same declaration with the same static or non-static status.

Any static field can be accessed only with the qualification of a class name, while an instance field can be accessed with the object reference qualification (see *Field Access Expression*).

If a field declaration is an implementation of one or more properties inherited from superinterfaces (see *Interface Inheritance*) then the types of the field and all properties must be the same. Otherwise, a compile-time error occurs.

```
1 // Two unrelated interfaces
2 interface B1 {}
3 interface B2 {}
4 // Interface which extends both of them
5 interface B3 extends B1, B2 {}
6 // Class which implements B3
```

(continues on next page)

(continued from previous page)

```
class BB3 implements B3 {}

interface II1 { f: B1 }
interface II2 { f: B2 } // Different property 'f' type as in II1
interface II3 { f: B1 } // The same property 'f' type as in II1

class CC1 implements II1, II2 {
    f: B1  = new BB3 /* compile-time error, field and all inherited properties
                        must be of the same type */
}
class CC2 implements II1, II3 {
    f: B3 = new BB3 /* compile-time error, field and all inherited properties
                       must be of the same type */
}
class CC3 implements II1, II3 {
    f: B1 = new BB3 // OK, correct properties implementation
}
```

9.6.1 Static and Instance Fields

There are two categories of class fields as follows:

- Static fields

  Static fields are declared with the modifier `static`. A static field is not part of a class instance. There is one copy of a static field irrespective of how many instances of the class (even if zero) are eventually created.

  Static fields are always accessed by using a qualified name notation wherever the class name is accessible (see *Accessible*).

  A compile-time error occurs if the name of a type parameter of the surrounding declaration is used as the type of a static field.

- Instance, or non-static fields

  Instance fields belong to each instance of the class. An instance field is created for, and associated with a newly-created instance of a class, or of its superclass. An instance field is accessible (see *Accessible*) via the instance name.

9.6.2 Readonly (Constant) Fields

A field with the modifier `readonly` is a *readonly field*. Changing the value of a readonly field after initialization is not allowed. Both static and non-static fields can be declared *readonly fields*.

9.6.3 Optional Fields

Optional field `f?: T = expr` effectively means that the type of `f``is ``T | undefined`. If an *initializer* is absent in a *field declaration*, then the default value `undefined` (see *Default Values for Types*) is used as the initial value of the field.

For example, the following two fields are actually defined the same way:

```
class C {
    f?: string
    g: string | undefined = undefined
}
```

9.6.4 Field Initialization

All fields except *Fields with Late Initialization* are initialized by using the default value (see *Default Values for Types*) or a field initializer (see below) if present. Otherwise, a field must be initialized as follows:

- Static field, by a static initialization block (see *Static Initialization*), or
- Instance field, by a class constructor (see *Constructor Declaration*).

Field initializer is an expression that is evaluated at compile time or runtime. The result of successful evaluation is assigned into the field. The semantics of field initializers is therefore similar to that of assignments (see *Assignment*). Each initializer expression evaluation and the subsequent assignment are only performed once.

The initializer of a non-static field declaration is evaluated at runtime. The assignment is performed each time an instance of the class is created (see *New Expressions*).

If the initializer expression uses `super` or `this` in any form, then a compile-time warning occurs.

If such use occurs then runtime error can occur as shown in the following examples:

```
class C {
    f0 = this // Compile-time warning as 'this' is used

    f1 = this.init_f1() // Compile-time warning as method of 'this' method is invoked

    init_f1 (): string {
       console.log (this.f1) // this.f1 was not yet initialized, a runtime error occurs
       return "a string field"
    }
}

class B {}
function foo (f: () => B) { return f() }
class A {
    field1 = foo(() => this.field2)
        // Compile-time warning as 'this' is used in the initializer code
        // At runtime the lambda call will lead to a runtime error as
        // this.field2 was not yet initialized

    field2 = new B
}
```

9.6.5 Fields with Late Initialization

Field with late initialization `f!: T = expr` effectively means that the type of the field `f` is `T | undefined`. However, the field behaves like a field of type `T` with any form of access.

Field with late initialization must be an *instance field*. Otherwise, a compile-time error occurs.

Field with late initialization cannot be of a *nullish type* (see *Nullish Types*). Otherwise, a compile-time error occurs.

As all other fields, a *field with late initialization* must be initialized before it is used for the first time. However, this field can be initialized *later* and not within a class declaration. Initialization of this field can be performed in a constructor (see *Constructor Declaration*), although it is not mandatory.

Field with late initialization cannot have *field initializers*, and can be neither `readonly` (see *Readonly (Constant) Fields*) nor `optional` (see *Optional Fields*). Otherwise, a compile-time error occurs. *Field with late initialization* must be initialized explicitly, even though its type has a *default value* which is ignored at the point of declaration.

Each time a *field with late initialization* is read, a field initialization check is performed. If the compiler identifies that a field is not initialized, then a compile-time error occurs. Otherwise, a check is performed at runtime, and if a non-initialized field is encountered, then a runtime error occurs:

```
class C {
    f!: string
}

let x = new C()
x.f = "aa"
console.log(x.f) // OK

let y = new C()
console.log(y.f) // runtime or compile-time error
```

Note

An initialization check on read slows down the access to a *field with late initialization*.

TypeScript uses the term *definite assignment assertion* for a notion similar to *late initialization*. However, ArkTS has a stricter read-check policy.

9.6.6 Override Fields

When extending a class, an instance field declared in a superclass can be overridden by a same-name, same-type field. The new declaration adds no new field to the class type but allows changing field initialization. Using the keyword `override` is not required.

Note

Implementing interface properties by fields is discussed in detail in *Implementing Required Interface Properties* and *Implementing Optional Interface Properties*.

A compile-time error occurs if:

- Field marked with the modifier `override` does not override a field from a superclass.
- Field declaration contains the modifier `static` along with the modifier `override`.
- Types of the overriding field and of the overridden field are different.
- *Access Modifiers* of the overriding field and of the overridden field are different.

```
class C {
    field: number = 1
    protected ff = 25
}
class D extends C {
    field: string = "aa"     // Compile-time error, type is not the same
    override no_field = 1224 // Compile-time error, no overridden field in the base class
    static override field: string = "aa" // Compile-time error, static cannot override
    ff = 66                  // Compile-time error, as access modifers are different
}
```

An overridden field must be initialized explicitly either by using an initializer or in a constructor. Otherwise, compile-time error occurs. Implicit initialization is not used, even though the type of the field has a default value (see *Default Values for Types*):

```
class C {
    f1: number = 1
    f2: Object = 2
}
class D extends C {
    f1: number = 7 // OK
    f2: Object     // OK, initialized in the constructor
    constructor () {
        super()
        this.f2 = "abc"
    }
}
class E extends C {
    f1: number  // Compile-time error, must be initialized explicitly
    f2: Object  // Compile-time error, must be initialized explicitly
}
```

Initializers of overridden fields are preserved for execution, and the initialization is normally performed in the context of *superclass* constructors.

```
class C {
    field: number = C.init()
    private static init() {
       console.log ("Field initialization in C")
       return 123
    }
```

(continues on next page)

(continued from previous page)

```
}
class D1 extends C {
    override field: number = 321 // field can be explicitly marked as overridden
}

class D2 extends D1 {
    field: number = D2.init_in_derived()
    private static init_in_derived() {
       console.log ("Field initialization in Derived")
       return 42
    }
}
console.log ((new D2()).field)
/* Output:
    Field initialization in C
    Field initialization in Derived
    42
*/
```

The term *same-type* in case of generic classes means that the type of a field in a derived class must be the same as in the base class instantiated with a parameter of the same type as the type of the field in the derived class.

The situation is represented in the example below:

```
class B<T> {
   f1: T
   f2: T
   constructor (v: T) { this.f1 = v; this.f2 = v }
}
class D<U, V> extends B<U>  {
   f1: U // Valid overriding as D extends B<U>, and type of f1 in B<U> is U
   f2: V // Compile-time error, wrong overriding
   constructor (v: U) {
       super (v)
   }
}
```

A compile-time error occurs if a field is not declared as `readonly` in a superclass, while an overriding field is marked as `readonly`:

```
class C {
    field = 1
}
class D extends C {
    readonly field = 2 // Compile-time error, wrong overriding
}
```

A compile-time error occurs if a field overrides a getter, a setter, or both in a superclass:

```
class C {
    get num(): number { return 42 }
    set num(x: number) {}
}
```

(continues on next page)

(continued from previous page)

```
class D extends C {
    num: number = 2 // Compile-time error, wrong overriding
}
```

Overriding a field by a single accessor or by both accessors causes a compile-time error as follows:

```
class C {
    num: number = 1
}
class D extends C {
    get num(): number  { return this.shadow } // Compile-time error, wrong overriding
    set num(x: number) { this.shadow = p }    // Compile-time error, wrong overriding
    private shadow: number = 123
}
```

9.7 Method Declarations

Methods declare executable code that can be called.

A *method* is defined by the following:

1. *Type parameter*, i.e., the declaration of any type parameter of the method member.
2. *Argument type*, i.e., the list of types of arguments applicable to the method member.
3. *Return type*, i.e., the return type of the method member.

The syntax of *class method declarations* is presented below:

```
classMethodDeclaration:
    methodModifier* identifier typeParameters? signature block?
    ;

methodModifier:
    'abstract'
    | 'static'
    | 'final'
    | 'override'
    | 'native'
    | 'async'
    ;
```

The identifier in a *class method declaration* defines the method name that can be used to refer to a method (see *Method Call Expression*).

Methods with the `final` modifier is an experimental feature discussed in detail in *Final Methods*.

A compile-time error occurs if:

- Method modifier appears more than once in a method declaration;
- Body of a class declaration declares a method but the name of that method is already used for a field in the same declaration.

A non-static method declared in a class can do the following:

- Implement a method inherited from a superinterface or superinterfaces;
- Override a method inherited from a superclass (see *Overriding in Classes*);
- Act as method declaration of a new method.

Class static methods never implement or override superclass or superinterface methods.

9.7.1 Static Methods

A method declared in a class with the modifier `static` is a *static method*.

A compile-time error occurs if:

- The method declaration contains another modifier (`abstract`, `final`, or `override`) along with the modifier `static`.
- The header or body of a class method includes the name of a type parameter of the surrounding declaration.

Static methods are always called without reference to a particular object. As a result, a compile-time error occurs if the keywords `this` or `super` are used inside a static method.

Static methods are not inherited from superclasses:

```
class Base {
    static foo() { console.log ("static foo() from Base") }
    static bar() { console.log ("static foo() from Base") }
}

class Derived extends Base {
    static foo(p: string) { console.log ("static foo() from Derived") }
}

Base.foo() // Output: static foo() from Base
Base.bar() // Output: static foo() from Base
Derived.bar()           // Compile-time error, there is no bar() in Derived
Derived.foo("a string") // Output: static foo() from Derived
Derived.foo()           // Compile-time error, there is no foo(p:string) in Derived
```

Note

Class static methods may access protected or private members of the same class type or derived one represented as parameters or local variables:

```
 class C {
   protected count1: number
   private   count2: number
   static getCount(c: C): number {
     const local_c = new C
     return c.count1 + c.count2 + local_c.count1 + local_c.count2 // OK
   }
   static handleDerived (b: B) {
```

```
        b.count1 + b.count2 // OK
    }
}
class B extends C {
    static dealWithProtected (b: B) {
        b.count1 // OK
        b.count2 // Compile-time error
    }
}

C.getCount (new C)      // will return the sum of counts
C.handleDerived (new B) // will work with protected and private fields
```

9.7.2 Instance Methods

A method that is not declared static is called *non-static method*, or *instance method*.

An instance method is always called with respect to an object that becomes the current object which the keyword `this` refers to during the execution of the method body.

9.7.3 Abstract Methods

An *abstract* method declaration introduces the method as a member along with its signature but without implementation. An abstract method is declared with the modifier `abstract` in the declaration.

Non-abstract methods can be referred to as *concrete methods*.

A compile-time error occurs if:

- An abstract method is declared private.
- The method declaration contains another modifier (`static`, `final`, `native`, or `async`) along with the modifier `abstract`.
- The declaration of an abstract method *m* does not appear directly within abstract class `A`.
- Any non-abstract subclass of `A` (see *Abstract Classes*) does not provide implementation for *m*.

An abstract method declaration provided by an abstract subclass can override another abstract method. An abstract method can also override non-abstract methods inherited from base classes or base interfaces as follows:

```
class C {
    foo() {}
}
interface I {
    foo() {} // default implementation
}
abstract class X extends C implements I {
```

(continues on next page)

(continued from previous page)

```
    abstract foo(): void /* Here abstract foo() overrides both foo()
                            coming from class C and interface I */
}
```

9.7.4 Async Methods

Async methods are discussed in *Asynchronous Methods*.

9.7.5 Overriding Methods

The `override` modifier indicates that an instance method in a superclass is overridden by the corresponding instance method from a subclass (see *Overriding*).

The usage of the modifier `override` is optional but strongly recommended as it makes the overriding explicit.

A compile-time error occurs if:

- Method marked with the modifier `override` overrides no method from a superclass.
- Method declaration contains modifier `static` along with the modifier `override`.

If the signature of an overridden method contains parameters with default values (see *Optional Parameters*), then the overriding method must always use the same default parameter values for the overridden method. Otherwise, a compile-time error occurs.

More details on overriding are provided in *Overriding in Classes* and *Overriding in Interfaces*.

9.7.6 Native Methods

Native methods are discussed in *Native Methods*.

9.7.7 Method Body

Method body is a block of code that implements a method. A semicolon or an empty body (i.e., no body at all) indicate the absence of implementation.

An abstract or native method must have an empty body.

In particular, a compile-time error occurs if:

- The body of an abstract or native method declaration is a block.

- The method declaration is neither abstract nor native, but its body is either empty or a semicolon.

The rules that apply to return statements in a method body are discussed in *return Statements*.

A compile-time error occurs if a method:

- Is declared to have a return type other than *void*, and
- Has no return statement on any execution path of its body.

9.7.8 Methods Returning `this`

A return type of an instance method can be `this`. It means that the return type is the class type to which the method belongs. It is the only place where the keyword `this` can be used as type annotation (see *Signatures* and *Return Type*).

The only result that is allowed to be returned from an instance method is `this`. There are two options to have `this` returned:

- Literally `return this`; or
- Return the result of any method that returns `this`.

A call to another method can return `this` or `this` statement:

```
class C {
    foo(): this {
        return this
    }
    bar(): this {
        return this.foo()
    }
}
```

The return type of an overridden method in a subclass must also be `this`:

```
class C {
    foo(): this {
        return this
    }
}

class D extends C {
    foo(): this {
        return this
    }
}

let x = new C().foo() // type of 'x' is 'C'
let y = new D().foo() // type of 'y' is 'D'
```

Otherwise, a compile-time error occurs.

9.8 Class Accessor Declarations

Class accessors are declared in the form of getters or setters, i.e., methods with predefined signatures that suport using field access syntax to call such methods.

Class accessors are often used instead of fields to add additional control for operations of getting or setting a field value.

The syntax of *class accessor declarations* is presented below:

```
classAccessorDeclaration:
    classAccessorModifier*
    ( 'get' identifier '(' ')' returnType? block?
    | 'set' identifier '(' parameter ')' block?
    )
    ;

classAccessorModifier:
    'abstract'
    | 'static'
    | 'final'
    | 'override'
    | 'native'
    ;
```

Accessor modifiers are a subset of method modifiers. The allowed accessor modifiers have exactly the same meaning as the corresponding method modifiers (see *Abstract Methods* for the modifier `abstract`, *Static Methods* for the modifier `static`, *Final Methods* for the modifier `final`, *Overriding Methods* for the modifier `override`, and *Native Methods* for the modifier `native`).

```
class Person {
  private _age: number = 0
  get age(): number { return this._age }
  set age(a: number) {
    if (a < 0) { throw new Error("wrong age") }
    this._age = a
  }
}
```

A *get-accessor* (*getter*) must have an explicit return type and no parameters, or no return type at all on condition it can be inferred from the getter body. A *set-accessor* (*setter*) must have a single parameter and no return type. The use of getters and setters looks the same as the use of fields. A compile-time error occurs if:

- Getters or setters are used as methods;
- Getter return type cannot be inferred from the getter body;
- *Set-accessor* (*setter*) has a single parameter that is optional (see *Optional Parameters*):

```
class Person {
  private _age: number = 0
  get age(): number { return this._age }
  set age(a: number) {
    if (a < 0) { throw new Error("wrong age") }
    this._age = a
  }
}
```

(continues on next page)

(continued from previous page)

```

let p = new Person()
p.age = 25        // setter is called
if (p.age > 30) { // getter is called
  // do something
}
p.age(17) // Compile-time error, setter is used as a method
let x = p.age() // Compile-time error, getter is used as a method

class X {
    set x (p?: Object) {} // Compile-time error, setter has optional parameter
}
```

If a getter has no return type specified, then the type is inferred as in *Return Type Inference*.

```
class Person {
  private _age: number = 0
  get age() { return this._age } // return type is inferred as number
}
```

A class can define a getter, a setter, or both with the same name. If both a getter and a setter with a particular name are defined, then both must have the same accessor modifiers. Otherwise, a compile-time error occurs.

Any accessor implementation can use a private or non-private field or fields to store the data as in the examples above and below.

```
class Person {
  forename: string = ""
  surname: string = ""
  get fullName(): string {
    return this.surname + " " + this.forename
  }
}
console.log (new Person().fullName)
```

A name of an accessor cannot be the same as that of a non-static field, or of a method of class or interface. Otherwise, a compile-time error occurs:

```
class Person {
  name: string = ""
  get name(): string { // Compile-time error, getter name clashes with the field name
      return this.name
  }
  set name(a_name: string) { // Compile-time error, setter name clashes with the field␣
↪name
      this.name = a_name
  }
}
```

In the process of inheriting and overriding (see *Overriding*), accessors behave as methods. The getter parameter type follows the covariance pattern, and the setter parameter type follows the contravariance pattern (see *Override-Compatible Signatures*):

```
class Base {
  get field(): Base { return new Base }
  set field(a_field: Derived) {}
}
class Derived extends Base {
  override get field(): Derived { return new Derived }
  override set field(a_field: Base) {}
}
function foo (base: Base) {
   base.field = new Derived // setter is called
   let b: Base = base.field // getter is called
}
foo (new Derived)
```

9.9 Constructor Declaration

Constructors are used to initialize objects that are instances of a class. A *constructor declaration* starts with the keyword `constructor`, and has optional name. In any other syntactical aspect, a constructor declaration is similar to a method declaration with no return type:

```
constructorDeclaration:
    'native'? 'constructor' identifier? parameters constructorBody?
    ;
```

The syntax and semantics of a constructor's formal parameters are identical to those of a method.

A constructor defined in supertype cannot be used directly in a *new expression* for a subtype, but is available for a call in all derived class constructors via `super` call (see *Explicit Constructor Call*).

```
class C {
    constructor (s: string) {}
}
class D extends C {
}

new D("aa") // Compile-time error, 'D' has default constructor only

class D1 extends C {
    constructor (n: number) {
        super("" + n) // OK
    }
}
```

Constructors are called by the following:

- Class instance creation expressions (see *New Expressions*); and
- Explicit constructor calls from other constructors (see *Constructor Body*).

Access to constructors is governed by access modifiers (see *Access Modifiers* and *Scopes*). Declaring a constructor inaccessible prevents class instantiation from using this constructor. If the only constructor is declared inaccessible,

then no class instance can be created.

A `native` constructor (an experimental feature described in *Native Constructors*) must have no *constructorBody*. Otherwise, a compile-time error occurs.

A non-`native` constructor must have *constructorBody*. Otherwise, a compile-time error occurs.

9.9.1 Constructor Body

Constructor body is a block of code that implements a constructor.

The syntax of *constructor body* is presented below:

```
constructorBody:
    '{' statement* '}'
    ;
```

The constructor body must provide correct initialization of new class instances. Constructors have two variations:

- *Primary constructor* that initializes instance own fields directly;
- *Secondary constructor* that uses another same-class constructor to initialize its instance fields.

The high-level sequence of a *primary constructor* body includes the following:

1. Optional arbitrary code that uses neither `this` nor `super`.
2. Call to a superconstructor (see *Explicit Constructor Call*) if a class has an extension clause (see *Class Extension Clause*). A compile-time error occurs if:
   - Call is not a root-level statement within a constructor;
   - Call argument uses `this` or `super`.
3. Implicit addition by the compiler of field initializers (if any) to be executed in the order they appear in a class body.
4. Code that neither uses `this` nor accesses fields through `this`, i.e., fields not explicity initialized in the body of a *primiary constructor* or during the step 3, are not read.
5. Arbitrary code after all fields are initialized as defined in **item 4**.

If the body has no call to a superconstructor (i.e., **item 2** above is omitted), then the compiler implicitly adds a superconstructor call with no arguments as the very first statement in the body. If the superconstructor requires a non-empty list of arguments, then a compile-time error occurs as represented below:

```
class C1 {
    constructor() {}
}
class D1 extends C1 {
    constructor () {
        // OK, call 'super()' is implicitly added
        /* other code here */
    }
}

class C2 {
```

(continues on next page)

(continued from previous page)

```
    constructor(n: number) {}
}
class D2 extends C2 {
    constructor () {
        // Compile-time error, call 'super()' cannot be implicitly added
        /*other code here*/
    }
}

class C3 {
    constructor(n?: number) {}
}
class D3 extends C3 {
    constructor () {
        // OK, call 'super()' is implicitly added
        /*other code here*/
    }
}
```

Field initialization referred in **item 4** above cannot be guaranteed at compile time in all possible cases. The following strategy is taken:

- If the compiler can detect that a non-initialized field is accessed during compilation, then a compile-time error occurs;
- If the limitation of **item 4** is violated, then a compile-time warning occurs;
- Otherwise, the execution-time behavior is determined by the implementation.

```
class Base {
  x: Object
  constructor() {
      this.x = new Object // Base object is fully initialized
      crash_this (this)   // A compiler may issue a compile-time error
  }
}
class Derived extends Base {
  y: Object
  constructor () {
      super() // mandatory call to base class constructor
      crash_this(this) // Compile-time warning as this.y is not initialized yet
                       // is guaranteed, however a compiler can issue a
                       // Compile-time error
      this.y = new Object
  }
}
function crash_this (b: Base) {
      if (b instanceof Derived) { // If b is of type Derived, then
            console.log ((b as Derived).y) // Access y field of Derived object
            // Depending on the compilation context, either the compiler reports
            // a compile-time error, or the runtime system is to detect the case
      }
}
```

The manner new expressions (see *New Expressions*) work together with a constructor body execution sequence is represented by the example below:

```
let count = 0
function trace (msg: string) {
   count++
   console.log (count + ": " + msg)
   return count
}

class C {
   C_field = trace("C fields initialization performed")
   constructor() {
      trace ("C constructor called")
   }
}
class B extends C {
   B_field = trace("B fields initialization performed")
   constructor() {
      super ()
      trace ("B constructor called")
   }
}
class A extends B {
   A_field = trace("A fields initialization performed")
   constructor(p: number) {
      super()
      trace ("A constructor called")
   }
}
new A (trace("constructor arguments evaluated"))

// The output
// "1: constructor arguments evaluated"
// "2: C fields initialization performed"
// "3: C constructor called "
// "4: B fields initialization performed"
// "5: B constructor called "
// "6: A fields initialization performed"
// "7: A constructor called "
```

Primary constructors are represented by the example below:

```
 class Point {
   x: number
   y: number
   constructor(x: number, y: number) {
     this.x = x
     this.y = y
   }
 }

 class ColoredPoint extends Point {
   static readonly WHITE = 0
```

(continues on next page)

(continued from previous page)

```
  static readonly BLACK = 1
  color: number
  constructor(x: number, y: number, color: number) {
    super(x, y) // calls base class constructor
    this.color = color
  }
}

class BWPoint extends ColoredPoint {
  constructor(x: number, y: number, black: boolean) {
    console.log ("Code that uses neither 'this' nor 'super'")
    super (x, y, black ? ColoredPoint.BLACK : ColoredPoint.WHITE)
    console.log ("Any code after that point")
  }
}
```

The high-level sequence of a *secondary constructor* body includes the following:

1. Optional arbitrary code that does not use `this` or `super`.
2. Mandatory call to another same-class constructor that uses the keyword `this` (see *Explicit Constructor Call*). A compile-time error occurs if:
   - The call is not a root-level statement within a constructor;
   - An argument of the call uses `this` or `super`.
3. Optional arbitrary code.

The example below represents *primary* and *secondary* constructors:

```
class Point {
  x: number
  y: number
  constructor(x: number, y: number) {
    this.x = x
    this.y = y
  }
}

class ColoredPoint extends Point {
  static readonly WHITE = 0
  static readonly BLACK = 1
  color: number

  // Primary constructor:
  constructor(x: number, y: number, color: number) {
    super(x, y) // Calls base class constructor as class has 'extends'
    this.color = color
  }
  // Secondary constructor:
  constructor (color: number) {
    this(0, 0, color) // Calls same class primary constructor
  }
}
```

A compile-time error occurs if a constructor calls itself, directly or indirectly through a series of one or more explicit constructor calls using `this`.

A constructor body looks like a method body (see *Method Body*), except for the semantics as described above. Explicit return of a value (see *return Statements*) is prohibited. On the opposite, a constructor body can use a return statement without an expression.

A constructor body can have no more than one call to the current class or direct superclass constructor. Otherwise, a compile-time error occurs.

9.9.2 Explicit Constructor Call

There are two kinds of *explicit constructor calls*:

- *Superclass constructor calls* (used to call a constructor from the direct superclass) that begin with the keyword `super`.
- *Other constructor calls* that begin with the keyword `this` (used to call another same-class constructor).

A compile-time error occurs if:

- *Superclass constructor call* refers to an inaccessible direct superclass constructor.
- *Explicit constructor call* is used as expression.

A compile-time error occurs if arguments of an explicit constructor call refer to one of the following:

- Any non-static field or instance method; or
- `this` or `super`.

```
class Base {
    constructor () {}
}
class Derived extends Base {
    constructor () {
        super()        // Call Base class constructor
    }
}
class Derived2 extends Base {
    constructor (x: number) {
        super()        // Call Base class constructor
    }
}
```

Semantic check for a `super` call is performed in accordance with *Compatibility of Call Arguments*.

9.9.3 Default Constructor

If a class contains no constructor declaration, then a default constructor is implicitly declared. This guarantees that every class effectively has at least one constructor. The form of a default constructor has the following:

- Default constructor has modifier `public` (see *Access Modifiers*).
- Default constructor has no parameters.

Default constructor body for any class except class `Object` contains:

- Call to a superclass parameterless constructor.
- Mandatory execution of field initializers (if any) in the order they appear in a class body.

Default constructor body for the class `Object` (see *Type Object*) is empty.

A compile-time error occurs if a class has a default constructor, but its superclass has no accessible constructor (see *Accessible*) without parameters.

```
// Class declarations with default constructors declared implicitly
class Base_no_ctor_declared {}
class Derived_no_ctor_declared extends Base_no_ctor_declared {}

// Example of an error case
class A {
    private constructor () {}
}
class B0 extends A {} // Compile-time error as default constructor for B0
                      // cannot call super() as it is private
class B1 extends A {
     constructor () {
         super ()   // Compile-time error as super() is private
     }
}
```

9.10 Inheritance

Class `C` inherits all non-static accessible members from its direct superclass and direct superinterfaces (see *Accessible*), and optionally overrides or implements some of the inherited members.

If `C` is not abstract, then it must implement all inherited abstract methods. The method of each inherited abstract method must be defined with *override-compatible* signatures (see *Override-Compatible Signatures*).

Semantic checks performed while overriding inherited methods and accessors are described in *Overriding in Classes*.

Semantic checks performed while overriding fields and implementing properties are described in the following sections:

- *Override Fields*,
- *Implementing Required Interface Properties*,
- *Implementing Optional Interface Properties*.

Constructors from the direct superclass of `C` are not subject of overriding because such constructors are not accessible (see *Accessible*) in `C` directly, and can only be called from a constructor of `C` (see *Constructor Body*).

CHAPTER

TEN

INTERFACES

An interface declaration declares an *interface type*, i.e., a reference type that:

- Includes properties and methods as its members;
- Has no instance variables (fields);
- Usually declares one or more methods;
- Allows otherwise unrelated classes to provide implementations for the methods, and so implement the interface.

Creating an instance of interface type is not possible.

An interface can be declared *direct extension* of one or more other interfaces. If so, the interface inherits all members from the interfaces it extends. Inherited members can be optionally overridden or hidden.

A class can be declared to *directly implement* one or more interfaces. Any instance of a class implements all methods specified by its interface(s). A class implements all interfaces that its direct superclasses and direct superinterfaces implement. Interface inheritance allows objects to support common behaviors without sharing a superclass.

The value of a variable declared *interface type* can be a reference to any instance of a class that implements the specified interface. However, it is not enough for a class to implement all methods of an interface. A class or one of its superclasses must be actually declared to implement an interface. Otherwise, the class is not considered to implement the interface.

The rules of subtyping are discussed in detail in *Subtyping for Non-Generic Classes and Interfaces* and *Subtyping for Generic Classes and Interfaces*.

10.1 Interface Declarations

Interface declaration specifies a new named reference type.

The syntax of *interface declarations* is presented below:

```
interfaceDeclaration:
    'interface' identifier typeParameters?
    interfaceExtendsClause? '{' interfaceMember* '}'
    ;

interfaceExtendsClause:
    'extends' interfaceTypeList
    ;
```

(continues on next page)

(continued from previous page)

```
interfaceTypeList:
    typeReference (',' typeReference)*
    ;
```

The *identifier* in an interface declaration specifies the interface name.

An interface declaration with `typeParameters` introduces a new generic interface (see *Generics*).

The scope of an interface declaration is defined in *Scopes*.

10.2 Superinterfaces and Subinterfaces

An interface declared with an `extends` clause extends all other named interfaces, and thus inherits all their members. Such other named interfaces are *direct superinterfaces* of a declared interface. A class that *implements* the declared interface also implements all interfaces that the interface *extends*.

A compile-time error occurs if:

- *typeReference* ` in the `extends` clause refers directly to, or is an alias of non-interface type.
- Interface type named by `typeReference` is not *Accessible*.
- Type arguments (see *Type Arguments*) of `typeReference` denote a parameterized type that is not well-formed (see *Generic Instantiations*).
- The `extends` graph has a cycle.

If an interface declaration (possibly generic) `I <`F_1 `, ..., `F_n`>` ($n \geq 0$) contains an `extends` clause, then the *direct superinterfaces* of the interface type `I <`F_1 `, ..., `F_n`>` are the types given in the `extends` clause of the declaration of `I`.

All *direct superinterfaces* of the parameterized interface type `I <`T_1 `, ..., `T_n`>` are types `J <`$U_1\theta$ `, ..., `$U_k\theta$`>`, if:

- T_i ($1 \leq i \leq n$) is the type of a generic interface declaration `I <`F_1 `, ..., `F_n`>` ($n > 0$);
- `J <`U_1 `, ..., `U_k`>` is a direct superinterface of `I <`F_1 `, ..., `F_n`>`; and
- θ is a substitution [F_1 `:=` T_1 `, ...,` F_n `:=` T_n].

The transitive closure of the direct superinterface relationship results in the *superinterface* relationship.

Interface *I* is a *subinterface* of *K* wherever *K* is a superinterface of *I*. Interface *K* is a superinterface of *I* if:

- *I* is a direct subinterface of *K*; or
- *K* is a superinterface of some interface *J* of which *I* is, in turn, a subinterface.

There is no single interface to which all interfaces are extensions (unlike class `Object` to which every class is an extension).

A compile-time error occurs if an interface depends on itself.

10.3 Interface Members

An *interface declaration* contains *interface members* which are either properties (see *Interface Properties*) or methods (see *Interface Method Declarations*).

The syntax of *interface member* is presented below:

```
interfaceMember
    : annotationUsage?
    ( interfaceProperty
    | interfaceMethodDeclaration
    | explicitInterfaceMethodOverload
    )
    ;
```

The scope of declaration of a member *m* that the interface type `I` declares or inherits is specified in *Scopes*.

The usage of annotations is discussed in *Using Annotations*.

Interface members include:

- Members declared explicitly in the interface declaration;
- Members inherited from a direct superinterface (see *Superinterfaces and Subinterfaces*).

A compile-time error occurs if the method explicitly declared by the interface has the same name as the `Object`'s `public` method.

```
interface I {
    toString (p: number): void // Compile-time error
    toString(): string { return "some string" } // Compile-time error
}
```

An interface inherits all members of the interfaces it extends (see *Interface Inheritance*).

A name in a declaration scope must be unique, i.e., the names of properties and methods of an interface type must not be the same (see *Interface Declarations*).

10.4 Interface Properties

Interface property is an *accessor* which can be declared in the form of a field declaration, or a getter or setter declaration, or getter and setter declarations.

The syntax of *interface property* is presented below:

```
interfaceProperty:
    'readonly'? identifier '?'? ':' type
    | 'get' identifier '(' ')' returnType
    | 'set' identifier '(' requiredParameter ')'
    ;
```

An interface property is a *required property* (see *Required Interface Properties*) if it is one of the following:

- Explicit *accessor*, i.e., a getter or a setter; or

- Form of a field that has no `'?'`.

Otherwise, it is an *optional property* (see *Optional Interface Properties*).

If `'?'` is used after the name of the property, then the property type is semantically equivalent to `type | undefined`.

```
interface I {
    property?: Type
}
// is the same as
interface I {
    property: Type | undefined
}
```

10.4.1 Required Interface Properties

A *required property* defined in the form of a field implicitly defines the following:

- Getter, if the property is marked as `readonly`;
- Otherwise, both a getter and a setter of the same name.

A type annotation for the field defines return type for the getter and type of parameter for the setter.

As a result, the following declarations have the same effect:

```
interface Style {
    color: string
}
// is the same as
interface Style {
    get color(): string
    set color(s: string)
}
```

Note

A *required property* defined in a form of accessors does not define any additional entities in the interface.

A class that implements an interface with properties can also use a field or an accessor notation (see *Implementing Required Interface Properties*, *Implementing Optional Interface Properties*).

10.4.2 Optional Interface Properties

An *optional property* can be defined in two forms:

- Short form `identifier '?' ':'  T`; or
- Explicit form `identifier ':'  T | undefined`.

In both cases, `identifier` has effective type `T | undefined`.

The *optional property* implicitly defines the following:

- A getter (if the property is marked as `readonly`);
- Otherwise, both a getter and a setter of the same name.

Accessors have implicitly defined bodies, in this aspect they are similar to *Default Interface Method Declarations*. However, ArkTS does not support explicitly defined accessors with bodies.

The following declaration:

```
interface I {
    num?: number
}
```

– implicitly declares two accessors:

```
interface I {
    get num(): number | undefined { return undefined }
    set num(x: number | undefined) { throw new InvalidStoreAccessError }
}
```

If the default setter is not overridden in a class that implements the interface, `InvalidStoreAccessError` is thrown at attempt to set value of an optional property. See also *Implementing Optional Interface Properties*.

10.5 Interface Method Declarations

An ordinary *interface method declaration* specifies the method name and signature, and is called *abstract*. Its implicit accessibility is `public`.

An interface method can have a body (see *Default Interface Method Declarations*) as an experimental feature.

The syntax of *interface method declaration* is presented below:

```
interfaceMethodDeclaration:
    identifier signature
    | interfaceDefaultMethodDeclaration
    ;
```

10.6 Interface Inheritance

Interface *I* inherits all properties and methods from its direct superinterfaces. Semantic checks are described in *Overriding in Interfaces* and *Implicit Method Overloading*.

> **Note**
>
> The semantic rules of methods apply to properties because any interface property implicitly defines a getter, a setter, or both.

Private methods defined in superinterfaces are not accessible (see *Accessible*) in the interface body.

A compile-time error occurs if interface I declares a `private` method m with a signature compatible with the instance method m' (see *Override-Compatible Signatures*) that has any access modifier in the superinterface of I.

The same scheme applies to properties and accessors:

```
interface I1 {
    prop1: number
    set prop2 (p: number)
    get prop3 (): number
}
interface I2 {
    prop1: number
    set prop2 (p: number)
    get prop3 (): number
}
interface I3 extends I1, I2 {}
// There is only one property prop1, prop2 and prop3 in I3

function foo (i3: I3) {
   i3.prop1 = 5 // Setter for prop1 is called
   i3.prop1     // Getter for prop1 is called
   i3.prop2 = 5 // Setter for prop2 is called
   i3.prop2     // Compile-time error as no getter for prop2
   i3.prop3 = 5 // Compile-time error as no getter for prop3
   i3.prop3     // Getter for prop3 is called
}
```

CHAPTER

ELEVEN

ENUMERATIONS

Enumeration type `enum` specifies a distinct user-defined type with an associated set of named readonly members that define its possible enumeration values.

The syntax of *enumeration declaration* is presented below:

```
enumDeclaration:
    'const'? 'enum' identifier enumBaseType? '{' enumMemberList? '}'
    ;

enumBaseType:
    ':' type
    ;

enumMemberList:
    enumMember (',' enumMember)* ','?
    ;

enumMember:
    identifier initializer?
    ;

initializer:
    '=' expression
    ;
```

If an *enumeration declaration* is prefixed with the keyword `const`, then a compile-time error occurs. This restriction is temporary, and the semantics of `const enum` is to be made available in the future versions of ArkTS.

All member names in an enumeration must be unique. Otherwise, a compile-time error occurs:

```
enum E3 { A = 5, A = 77 }      // Compile-time error
enum E4 { A = 5, B = 5 }       // OK, values can be the same
```

Empty `enum` is supported as a corner case for compatibility with TypeScript.

```
enum Empty {} // OK
```

An enumeration member has a value that can be set explicitly by an initializer expression or implicitly (see *Initialization of Enumeration Members*).

Qualification by type name is mandatory to access the enumeration member, except initializer expression:

```
enum Color {
    Red,
    Green,
    Blue,
    Default = Red // Qualification can be omitted
}
let c: Color = Color.Red // Qualification is mandatory
```

The following operators can be applied to operands of enumeration types:

- Relational operators (see *Enumeration Relational Operators*);
- Equality operators (see *Equality Expressions*).

Implicit conversions (see *Enumeration to Numeric Type Conversion*, *Enumeration to string Type Conversion*) are used to convert a value of enumeration type to numeric types or type `string` depends on the enumeration base type:

```
enum Color { Red, Green, Blue }
let x: number = Color.Red + 1 // OK, implicit conversion to 'int'

enum T { A = "hello", B = "Bye" }

let s: string = T.A // OK, implicit conversion to 'string'
```

If enumeration type is exported, then all enumeration members are exported along with the mandatory qualification.

For example, if *Color* is exported, then all members like `Color.Red` are exported along with the mandatory qualification `Color`.

11.1 Enumeration Base Type

Each enumeration has an *enumeration base type*. A base type can be set explicitly (see *Enumeration with Explicit Base Type*) or inferred from initializers as follows:

- If at least one member does not have an initializer, the inferred type is `int`. Types of all initializers must be assignable to type `int` (see *Assignability*). Otherwise, a compile-time error occurs.
- If types of all initializer expressions are assignable to type `int` then the inferred type is `int`;
- If types of all initializer expressions are assignable to type `string` then the inferred type is `string`;
- Otherwise, a compile-time error occurs.

It is discussed in detail in *Initialization of Enumeration Members*.

Cases with base type inference are represented in the example below:

```
enum E1 { A, B }     // OK, base type is 'int'
enum E2 { A = 5, B } // OK, base type is 'int'

enum E3 { A, B = "hello"}     // Compile-time error, base type cannot be inferred
enum E4 { A = 5, B = "hello"} // Compile-time error, base type cannot be inferred
```

11.2 Enumeration with Explicit Base Type

Enumeration base type can be set explicitly as represented in the example below:

```
enum DoubleEnum: double { A = 0.0, B = 1, C = 3.14159 }
enum ByteEnum: byte { A = 0, B = 1, C = 3 }
```

A compile-time error occurs in the following situations:

- *Explicit type* is different from any numeric or string type.
- Type of explicit member initializer is not assignable (see *Assignability*) to the base type;
- Enumeration member has no explicit initializer and the base type is not an integer type (see *Initialization of Enumeration Members*).

```
enum DoubleEnum: double { A = 0.0, B = 1, C = 3.14159 } // OK
enum LongEnum: long { A = 0, B = 1, C = 3 } // OK

enum Wrong1: double { A, B, C } // Compile-time error, must be explicitly initialized
enum Wrong2: long { A = 1, B = "abc" } // Compile-time error, not assignable to 'long'
```

11.3 Initialization of Enumeration Members

An enumeration member value can be set explicitly by an initializer expression. An initializer expression can be omitted if an *enumeration base type* is an integer type. Otherwise, a compile-time error occurs:

```
enum E1 { A, B, C }    // OK, base type is 'int'
enum E2: long { K, L } // OK, base type is 'long'

enum E3 { A = "a", B }   // Compile-time error, wrong base type
enum E4: string { A, B } // Compile-time error, base type is not an integer type
enum E5: double { C, D } // Compile-time error, base type is not an integer type
```

If initializers for all members are omitted, then the first member is assigned the value zero. The other member is assigned the value of the immediately preceding member plus one.

If some but not all members have their values set explicitly, then the values of the members are set by the following rules:

- If an initializer is not a *constant expression* (see *Constant Expressions*), then all subsequent members must be explicitly initialized. Otherwise, a compile-time error occurs.
- Member which is the first and has no explicit initializer gets the zero value.
- Member with an explicit initializer has that explicit value.
- Member that is not the first and has no explicit initializer takes the value of the immediately preceding constant plus one.

In the example below, the value of `Red` is 0, of `Blue`, 5, and of `Green`, 6:

```
enum Color { Red, Blue = 5, Green }
```

The example below illustrates the requirement to have explicit initializers after non-constant initializer:

```
function foo() { return 1 }

enum Wrong {
    A,
    B = foo(),
    C // Compile-time error, must have explicit initializer
}
```

11.4 Enumeration Methods

The name of enumeration type can be indexed by the value of this enumeration type to get the name of the member:

```
enum Color { Red, Green = 10, Blue }
let c: Color = Color.Green
console.log(Color[c]) // Prints: Green
```

If several enumeration members have the same value, then the textually last member has the priority:

```
enum E { One = 1, one = 1, oNe = 1 }
console.log(E[E.fromValue(1)]) // Prints: oNe
console.log(E.fromValue(1).getName()) // Prints: oNe
```

Several static methods are available for each enumeration class type as follows:

- Method `static values()` returns an array of enumeration members in the order of declaration.
- Method `static getValueOf(name: string)` returns an enumeration member with the given name, or throws an error if no member with such name exists.
- Method `static fromValue(value: T)`, where `T` is the base type of the enumeration, returns an enumeration member with a given value, or throws an error if no member has such a value.

```
enum Color { Red, Green, Blue = 5 }
let colors = Color.values()
// colors[0] is the same as Color.Red

let red = Color.getValueOf("Red")

Color.fromValue(5) // OK, returns Color.Blue
Color.fromValue(6) // Throws runtime error
```

Methods for instances of an enumeration type are as follows:

- Method `toString()` converts a value of enumeration member to type `string`:

```
enum Color { Red, Green = 10, Blue }
let c: Color = Color.Green
console.log(c.toString()) // Prints: 10
```

- Method `valueOf()` returns a numeric or `string` value of an enumeration member depending on the enumeration base type.
- Method `getName()` returns the name of an enumeration member.

```
enum Color { Red, Green = 10, Blue }
let c: Color = Color.Green
console.log(c.valueOf()) // Prints 10
console.log(c.getName()) // Prints Green
```

Note

Methods `c.toString()` and `c.valueOf().toString()` return the same value.

CHAPTER

TWELVE

ERROR HANDLING

ArkTS is designed to provide first-class support in responding to, and recovering from different error situations in a program. Normal program execution can be interrupted by the occurrence of situations of two kinds:

- Runtime errors (e.g., null pointer dereferencing, array bounds checking, or division by zero);
- Operation completion failures (e.g., the task of reading and processing data from a file on disk can fail if the file does not exist on a specified path, read permissions are not available, or else).

The term *error* in this Specification denotes all kinds of error situations.

12.1 Errors

Error is the base class of all error situations. Defining a new error class is normally not required because essential error classes for various cases (e.g., `RangeError`) are defined in the standard library (see *Standard Library*).

However, a developer can handle a new error situation by using `Error` class itself, or by a subclass of `Error`. An example of error handling is provided below:

```
class UnknownError extends Error { // user-defined error class
   error: Error
   constructor (error: Error) {
      super()
      this.error = error
   }
}

function get_array_element<T>(array: T[], index: int): T|undefined {
    try {
      return array[index] // RangeError if index < 0 or index >= array.length
    }
    catch (error) {
      if (error instanceof RangeError) // invalid index detected
         return undefined
      throw new UnknownError (error) // unknown error occurred
    }
}

let arr = [1, 2, 3]
```

(continues on next page)

(continued from previous page)

```
let val = get_array_element(arr, -3) // RangeError: index -3 < 0

console.log(val) // Output: undefined
```

In most cases, errors are raised by the ArkTS runtime system, or by the standard library (see *Standard Library*) code.

New error situations can be created and raised by `throw` statements (see *throw Statements*) .

Errors are handled by using `try` statements (see *try Statements*).

Note

Some errors cannot be recovered.

```
function handleAll(
  actions : () => void,
  handling_actions : () => void)
{
  try {
    actions()
  }
  catch (x) { // Type of x is Error
      handling_actions()
  }
}
```

CHAPTER

THIRTEEN

NAMESPACES AND MODULES

Programs in ArkTS are structured as sequences of elements ready for compilation in the form of *modules* (see *Module Declarations*). A module is a specific form of a namespace called *top-level namespace*. A module can contain nested namespaces (see *Namespace Declarations*).

13.1 Module Declarations

Each module creates its own scope (see *Scopes*). Variables, functions, classes, interfaces, or other declarations of a module are only accessible (see *Accessible*) within such a scope if not exported explicitly. Types of all exported entities must be set explicitly (see details in *Export Directives*).

A variable, function, class, interface, or other declarations exported from a module must be imported first by the module that is to use them.

> **Note**
>
> Only exported declarations are available for the third party tools and programs written in other programming languages.

The modules can consist of one or more files (see *Multifile Module*).

All *modules* are stored in a file system or a database (see *Modules in Host System*).

A *module* can optionally consist of the following parts:

1. The module header that defines a module name;
2. Import directives allow using declarations imported into the current module within this module;
3. Top-level declarations;
4. Top-level statements; and
5. Re-export directives.

The syntax of *module* is presented below:

```
moduleDeclaration:
    moduleHeader? importDirective* (topDeclaration | topLevelStatements |␣
↪exportDirective)*
    ;
```

The accessibility of a module by import in other modules is determined by the build system.

Every module can directly use a set of exported entities defined in the standard library (see *Standard Library Usage*).

```
// Hello, world! module
function main() {
  console.log("Hello, world!") // console is defined in the standard library
}
```

If a module has at least one top-level ambient declaration (see *Ambient Declarations*), then all other declarations must be ambient, and no top-level statement must be present (see *Top-Level Statements*). Otherwise, a compile-time error occurs.

```
declare let x: number
function main() {}
// Compile-time error, ambient and non-ambient declarations are mixed
```

13.2 Module Header

Module header defines the optional modifier `export` and a *module name*.

The syntax of *module header* is presented below:

```
moduleHeader:
    'declare'? 'export'? 'module' moduleName
    ;

moduleName:
    StringLiteral
    ;
```

If a *module header* has the `declare` modifier than the whole module is ambient. Otherwise, a compile-time error occurs.

The `export` modifier is used by the build system to make a module accessible (see *Accessible*) in other modules by import (see *Import Directives*).

The usage of *module header* is represented in the example below:

```
export module "x"
import {A} from "some module"
export class B extends A {}
```

13.3 Namespace Declarations

Namespace declaration introduces a named container of entities with distinguishable names. Each namespace creates its own scope (see *Scopes*). Variables, functions, classes, interfaces, or other declarations of a namespace are only accessible (see *Accessible*) within this scope if not exported explicitly. Any use of exported entities is to be qualified with the name of a namespace.

The syntax of *namespace declarations* is presented below:

```
namespaceDeclaration:
    'namespace' qualifiedName
    '{' (topDeclaration | topLevelStatements | staticBlock | exportDirective)* '}'
    ;
```

Namespaces can have *top-level statements* or a *static block* which constitute a namespace initializer. The initializer is executed only if at least one of the exported namespace members is used in the program (see *Static Initialization* for detail).

Static block is to be deprecated in one of the future versions of ArkTS, using *top-level statements* is recommended instead. Only one *static block* is allowed in a namespace. Otherwise, a compile-time error occurs.

The usage of namespaces is represented in the example below:

```
namespace NS1 {
    export function foo(): void {  }
    export let variable: int = 1234
    export const constant: int = 1234
    export let someVar: string

    // Will be executed before any use of NS1 members
    someVar = "some string"
    console.log("Init for NS1 done")
    export function bar(): void {}
}

namespace NS2 {
    export const constant:int = 1
    // Will never be executed since NS2 members are never used
    console.log("Init for NS2 done")
    export function bar(): void {}
}

export function bar(): void {}  // That is a different bar()

if (NS1.variable == NS1.constant) {
    NS1.variable = 4321
}
NS1.bar()  // namespace bar() is called
bar()      // top-level bar() is called
```

Note

An exported namespace entity can be used in the form of a *qualifiedName* outside a namespace in the same module. Any namespace entity can be and typically is used inside a namespace without qualification, i.e., without a

namespace name. A *qualifiedName* inside a namespace can be used for a namespace entity only when the entity is exported. Using a *qualifiedName* for non-exported entity both inside and outside a namespace causes a compile-time error:

```
namespace NS {
    export let a: number = 1
    let b = 2

    export function foo(): void {
        let v: number
        v = a // OK, no qualification
        v = NS.a // OK, `a` exported
    }

    export function bar(): void {
        let v: number
        v = b  // OK, no qualification
        v = NS.b // Compile-time error, `b` not exported
    }
}

NS.a = 1 // OK,  `NS.a` exported
NS.b = 1 // Compile-time error, `NS.b` not exported
```

Note

A namespace must be exported to be used in another module:

```
// File1
namespace Space1 {
    export function foo(): void { ... }
    export let variable: int = 1234
    export const constant: int = 1234
}
export namespace Space2 {
    export function foo(p: number): void { ... }
    export let variable: int = "1234"
}

// File2
import {Space2 as Space1} from "File1"

// Compile-time error - there is no variable or constant called 'constant'
if (Space1.variable == Space1.constant) {
     // Compile-time error - incorrect assignment as type 'number'
     // is not compatible with type 'string'
    Space1.variable = 4321
}
Space1.foo()     // Compile-time error - there is no function 'foo()'
Space1.foo(1234) // OK
```

Note

Embedded namespaces are allowed:

```
namespace ExternalSpace {
    export function foo(): void { ... }
    export let variable: number = 1234
    export namespace EmbeddedSpace {
        export const constant: int = 1234
    }
}

if (ExternalSpace.variable == ExternalSpace.EmbeddedSpace.constant) {
    ExternalSpace.variable = 4321
}
```

Note

Namespaces with identical namespace names in a single module merge their exported declarations into a single namespace. A duplication causes a compile-time error. Exported and non-exported declarations with the same name are also considered a compile-time error. Only one of the merging namespaces can have an initializer. Otherwise, a compile-time error occurs.

```
// One source file
namespace A {
    export function foo(): void { console.log ("1st A.foo() exported") }
    function bar(): void {  }
    export namespace C {
        export function too(): void { console.log ("1st A.C.too() exported") }
    }
}

namespace B {  }

namespace A {
    export function goo(): void {
        A.foo() // calls exported foo()
        foo()   /* calls exported foo() as well as all A namespace
                   declarations are merged into one */
        A.C.moo()
    }
    //export function foo(): void {  }
    // Compile-time error as foo() was already defined

    // function foo() { console.log ("2nd A.foo() non-exported") }
    // Compile-time error as foo() was already defined as exported
}

namespace A.C {
    export function moo(): void {
```

(continues on next page)

(continued from previous page)

```
        too() // too()  accessible when namespace C and too() are both exported
        A.C.too()
    }
}

A.goo()

// File
namespace A {
    export function foo(): void { ... }
    export function bar(): void { ... }
}

namespace A {
    function goo() { bar() }  // exported bar() is accessible in the same namespace
    export function foo(): void { ... }  // Compile-time error as foo() was already defined
}
```

Note

A namespace name can be a qualified name. It is a shortcut notation of embedded namespaces as represented below:

```
namespace A.B {
    /*some declarations*/
}
```

The code above is a shortcut to the following code:

```
namespace A {
    export namespace B {
      /*some declarations*/
    }
}
```

This code is illustrative of the use of declarations in the case below:

```
namespace A.B.C {
    export function foo(): void { ... }
}

A.B.C.foo() // Valid function call, as 'B' and 'C' are implicitly exported
```

Where a namespace merges with qualified names, all qualification levels are merged. A compile-time error occurs in the case of a duplication.

```
namespace A {
    function foo() { ... } // #1
    export namespace B {
       export function foo(): void { ... } // #2
    }
}

namespace A.B {
    export function foo(): void { ... } // #3
}
// Declarations of functions #2 and #3 lead to a compile-time error,
//   duplicated declaration of function A.B.foo()

// While function foo() in namespace A is a valid declaration
```

If an ambient namespace (see *Ambient Namespace Declarations*) is defined in a module (see *Module Declarations*), then all ambient namespace declarations are accessible across all declarations and top-level statements of the module.

```
declare namespace A {
    function foo(): void
    type X = Array<number>
}

A.foo() // Valid function call, as 'foo' is accessible for top-level statements
function foo () {
    A.foo() // Valid function call, as 'foo' is accessible here as well
}
class C {
    method () {
        A.foo() // Valid function call, as 'foo' is accessible here too
        let x: A.X = [] // Type A.X can be used
    }
}
```

13.4 Import Directives

Import directives make entities exported from other modules (see *Module Declarations*) available for use in the current module by using different binding forms. These directives have no effect during the program execution.

An import declaration has the following two parts:

- Import path that determines from what module to import;
- Import bindings that define what entities, and in what form (either qualified or unqualified) the current module can use.

The syntax of *import directives* is presented below:

```
importDirective:
    'import' 'type'? bindings 'from' importPath
    ;

bindings:
    defaultBinding
    | (defaultBinding ',')? allBinding
    | (defaultBinding ',')? selectiveBindings
;

allBinding:
    '*' bindingAlias
    ;

bindingAlias:
```

(continues on next page)

(continued from previous page)

```
    'as' identifier
    ;

defaultBinding:
    'type'? identifier
    ;

selectiveBindings:
    nameBinding (',' nameBinding)*
    ;

nameBinding:
    'type'? identifier bindingAlias?
    | 'default' 'as' identifier
    ;

importPath:
    StringLiteral
    ;
```

Each binding adds an entity or entities to the scope of a module (see *Scopes*). Any entity so added must be distinguishable in the declaration scope (see *Declarations*).

Import with `type` modifier is discussed in *Import Type Directive*.

A compile-time error occurs if:

- A non-import directive, declaration, or statement precedes an `import` directive;
- Entity added to the scope of a module by a binding is not distinguishable;
- Module imports itself directly, and `importPath` refers to a file in which the current module is stored; or
- `import type` is used while `type` is also used by one of bindings.

A compile-time warning occurs if the `importPath` refers to a file with no ArkTS code, where the grammar start symbol corresponds to an empty one.

13.4.1 Bind All with Qualified Access

Import binding `* as A` binds the single named entity *A* to the declaration scope of the current module.

A qualified name consisting of *A* and the name of entity `A.name` is used to access any entity exported from the module as defined by the *import path*.

Import	Usage
`import * as Math from "..."`	`let x = Math.sin(1.0)`

This form of import is recommended because it simplifies the reading and understanding of the source code when all exported entities are prefixed with the name of the imported module.

13.4.2 Default Import Binding

Default import binding allows importing a declaration exported from a module as default export. Knowing the actual name of a declaration is not required as the new name is given at importing. A compile-time error occurs if another form of import is used to import an entity initially exported as default.

There are two forms of *default import binding*:

- Single identifier;
- Special form of selective import with the keyword `default`.

```
// Module 1
import DefaultExportedItemBindedName from "Module 2"
import {default as DefaultExportedItemNewName} from "Module 2"
function foo () {
  let v1 = new DefaultExportedItemBindedName()
  // instance of class 'SomeClass' to be created here
  let v2 = new DefaultExportedItemNewName()
  // instance of class 'SomeClass' to be created here
}

// Module 2
export default class SomeClass {}

// Module 3 - the same semantics as in Module 2
class SomeClass {}
export default SomeClass

// Module 4
export {default} from "Module 2" // Module 4 re-exports default export of Module 2
export {default as SomeClassNewName} from "Module 2"
  // Module 4 re-exports default export of Module 2 as a new exported name
```

13.4.3 Selective Binding

Selective binding allows to bind an entity exported as *identifier*, or an entity exported by default (see *Default Import Binding*).

Binding with *identifier* binds an exported entity with the name *identifier* to the declaration scope of the current module. If no *binding alias* is present, then the entity is added to the declaration scope under the original name. Otherwise, the identifier specified in *binding alias* is used. In the latter case, the bounded entity is no longer accessible (see *Accessible*) under the original name.

If an *identifier* refers to an *overloaded function* (see *Overloading*), then all accessible overloaded functions are imported. An exported *explicit function overload* (see *Explicit Function Overload*) is imported by its own exported name.

```
// File1
export function foo(p: number): void {} // #1
export function foo(p: string): void {} // #2
export function fooString(p: string): void {}
export function fooBoolean(p: Boolean): void {}
export overload bar {fooBoolean, fooString}

function foo() {} // #3

// File2
import {foo} from "File1"  // all exported 'foo' are imported
import {bar} from "File1"  // exported explicit overload is imported by its own name
foo(5)          // #1 is called
foo("a string") // #2 is called
bar(true)       // fooBoolean is called
bar("a string") // fooString is called
foo()           // Compile-time error, as #3 is not exported
```

Selective binding that uses exported entities is represented in the examples below:

```
export const PI: number = 3.14
export function sin(d: number): number {}
```

Note

The import path of the module is irrelevant and replaced for `'...'` in the examples below:

Import	Usage
`import {sin} from "..."`	`let x = sin(1.0)` `let f: float = 1.0`
`import {sin as Sine} from "..."`	`let x = Sine(1.0) // OK` `let y = sin(1.0) /* Error 'sin' is not accessible */`

A single import directive can list several names as follows:

Import	Usage
`import {sin, PI} from "..."`	`let x = sin(PI)`
`import {sin as Sine, PI} from "..."`	`let x = Sine(PI)`

Complex cases with several bindings mixed on one import path are discussed below in *Several Bindings for One Import Path*.

13.4.4 Import Type Directive

An import directive can have a `type` modifier exclusively for a better syntactic compatibility with TypeScript (also see *Export Type Directive*). ArkTS supports no additional semantic checks for entities imported by using *import type* directives.

The semantic checks performed by the compiler in TypeScript but not in ArkTS are represented by the following code:

```
// File module.ets
console.log ("Module initialization code")

export class Class1 {/*body*/}

class Class2 {}
export type {Class2}

// MainProgram.ets

import {Class1} from "./module.ets"
import type {Class2} from "./module.ets"

let c1 = new Class1() // OK
let c2 = new Class2() // Compile-time error in |TS|, OK in |LANG|
```

Another form of *type import* is used when `type` is attached to a name binding. This allows mixing general import and `type` import.

```
// File module.ets
console.log ("Module initialization code")

class Class1 {/*body*/}
class Class2 {}
export {Class1, type Class2}
```

(continues on next page)

(continued from previous page)

```
// MainProgram.ets

import {Class1, type Class2 } from "./module.ets"

let c1 = new Class1() // OK
let c2 = new Class2() // Compile-time error in |TS|, OK in |LANG|
```

13.4.5 Import Path

Import path is a string literal that determines where and how an imported module is to be searched for.

Import path can include the following:

- Initial dot `'.'` or two dots `'..'` followed by the slash character `'/'`.
- One or more path components (the subset of characters and case sensitivity of path components must follow the path rules of a host file system).
- Slash characters separating components of the path.

The slash character `'/'` is used in import paths irrespective of the host system. The backslash character is not used in this context.

In most file systems, an import path looks like a file path. *Relative* (see below) and *non-relative* import paths have different *resolutions* that map the import path to a file path of the host system.

The compiler uses its own algorithm to locate a module source that processes the import path. If the import path specifies no file extension, then the compiler can append some according to its own rules and priorities. If the import path refers to a folder, then the way to handle the case is determined by the actual compiler. If the compiler cannot locate a module source definitely, then a compile-time error occurs.

A *relative import path* starts with `'./'` or `'../'`. Examples of relative paths are presented below:

```
"./components/entry"
"../constants/http"
```

Resolving *relative import* is relative to the importing file. *Relative import* is used on modules to maintain their relative location.

```
import * as Utils from "./mytreeutils"
```

Other import paths are *non-relative*.

Resolving a *non-relative path* depends on the compilation environment. The definition of the compiler environment can be particularly provided in a configuration file or environment variables.

The *base URL* setting is used to resolve a path that starts with `'/'`. *Path mapping* is used in all other cases. Resolution details depend on the implementation. For example, the compilation configuration file can contain the following lines:

```
"baseUrl": "/home/project",
"paths": {
    "std": "/arkts/stdlib"
}
```

In the example above, `/net/http` is resolved to `/home/project/net/http`, and `std/components/treemap` to `/arkts/stdlib/components/treemap`.

File name, placement, and format are implementation-specific.

If the above configuration is in effect, the first path maps directly to file system after applying `baseUrl`, while `std` in the second path is replaced for `/arkts/stdlib`. Examples of non-relative paths are presented below.

```
"/net/http"
"std/components/treemap"
```

13.4.6 Several Bindings for One Import Path

The same bound entities can use the following:

- Several import bindings,
- One import directive, or several import directives with the same import path:

In one import directive	`import {sin, cos} from "..."`
In several import directives	`import {sin} from "..."` `import {cos} from "..."`

No conflict occurs in the above example, because the import bindings define disjoint sets of names.

The order of import bindings in an import declaration has no influence on the outcome of the import.

The rules below prescribe what names must be used to add bound entities to the declaration scope of the current module if multiple bindings are applied to a single name:

Case	Sample	Rule
A name is explicitly used without an alias in several bindings.	`import {sin, sin} from "..."`	OK. The compile-time warning is recommended.
A name is used explicitly without alias in one binding.	`import {sin} from "..."`	OK. No warning.
A name is explicitly used without alias, and implicitly with alias.	`import {sin} from "..."` `import * as M from "..."`	OK. Both the name and qualified name can be used: sin and M.sin are accessible.
A name is explicitly used with alias.	`import {sin as Sine} from "..."`	OK. Only alias is accessible for the name, but not the original name: • Sine is accessible; • sin is not accessible.
A name is explicitly used with alias, and implicitly with alias.	`import {sin as Sine} from "..."` `import * as M from "..."`	OK. Both options can be used: • Sine is accessible; • M.sin is accessible.
A name is explicitly used with alias several times.	`import {sin as Sine, sin as SIN} from "..."`	OK. Both aliases are accessible. But warning can be displayed.

13.5 Top-Level Declarations

Top-level declarations declare top-level types (`class`, `interface`, or `enum` see *Type Declarations*), top-level variables (see *Variable Declarations*), constants (see *Constant Declarations*), functions (see *Function Declarations*, overloads (see *Explicit Function Overload*), namespaces (see *Namespace Declarations*), or other declarations (see *Ambient Declarations*, *Annotations*, *Accessor Declarations*, *Functions with Receiver*). Top-level declarations can be exported.

The syntax of *top-level declarations* is presented below:

```
topDeclaration:
    ('export' 'default'?)?
    annotationUsage?
    ( typeDeclaration
    | variableDeclarations
```

(continues on next page)

(continued from previous page)

```
    | constantDeclarations
    | functionDeclaration
    | explicitFunctionOverload
    | namespaceDeclaration
    | ambientDeclaration
    | annotationDeclaration
    | accessorDeclaration
    | functionWithReceiverDeclaration
   )
   ;
```

```
export let x: number[], y: number
```

The usage of annotations is discussed in *Using Annotations*.

13.6 Exported Declarations

Top-level declarations can use modifier `export` that make the declarations accessible (see *Accessible*) in other modules by using import (see *Import Directives*). The same result can be achieved by using an export directive (see *Export Directives*) for a top-level declaration. Declarations that are not exported as mentioned above can be used only inside the module they are declared in.

```
export class Point {}
export let Origin: Point = new Point(0, 0)
export function Distance(p1: Point, p2: Point): number {
  // ...
}
```

If a declaration is exported with the name of another exported declaration, then a compile-time error occurs.

The situation where an *export directive* uses *selectiveBindings* or *bindingAlias* to give a new name to a declaration, and the new name clashes with the name of another exported declaration is represented in the example below:

```
export function foo(): void {}
function bar(): void {}
export {bar as foo} // Compile-time error, entity named 'foo' is already exported
```

The same error occurs when *Re-Export Directive* is used as represented in the example below:

```
// file1.ets
export class A {}

// file2.ets
export class A {}

// Another file
export * from "./file1"
export * from "./file2" // Compile-time error, entity named 'A' is already exported
```

In addition, only one top-level declaration can be exported by using the default export directive. It allows specifying no declared name when importing (see *Default Import Binding* for details). A compile-time error occurs if more than one top-level declaration is marked as `default`.

```
export default let PI: number = 3.141592653589
```

Another supported form of *export default* is using an expression as export default target. This export directive effectively means that an anonymous constant variable is created with a value equal to the value of the expression evaluation result. The export can be imported only by providing a name for the constant variable that is exported by using this export directive. Otherwise, a compile-time error occurs.

```
// File1
class A {
  foo () {}
}
export default new A

// File2
import {default as a} from "File1"

a.foo()  // Calling method foo() of class A where 'a' is an instance of type A
a = new A // Compile-time error as 'a' is a constant variable

// File3
import * as a from "File1" /* compile-time error, such form of import
                              cannot be used for the default export */
```

ArkTS forbids any exported declaration which, when imported into a code, causes a module to access an unexported entity. It applies to both global and namespace declarations. Types of exported functions, variables, constants, public and protected members of classes, default interface methods, interface getters and setters must be set explicitly where applicable.

Note

The statement *types of exported functions, ..., methods, ...* above means not only the return type but the entire signature of an entity, including, where applicable, types of parameters. E.g., a *setter* has no return type (not even `void`), but the type of its parameter must be exported explicitly.

Any entity declared in the current module and used in an accessible part of an exported declaration must be either directly available in ArkTS (e.g., built-in type), or also be exported. Otherwise, a compile-time error occurs.

Note

By an *accessible part of an exported declaration* we mean a set of entities which can be used in another module after importing the declaration. For example, public and protected members of an exported class belong to that set while the private members do not belong.

Note

The above requirements of explicit type and explicit export are also applied to *Ambient Declarations*.

Here is a number of examples which violate the above rules and therefore trigger a compile-time error,

- Exported constants and variables, or non-private fields of exported classes do not have explicit types. Exported functions, or methods of exported interfaces, or non-private methods of exported classes do not have explicit return types. Exported getters (see *Accessor Declarations*) at the top level or in a namespace do not have an explicit return types;
- An exported function or an accessor uses an unexported entity in its signature. An exported variable, or constant uses an unexported entity as a type. Note, that use of an unexported entity as a default for an *Optional Parameters* is allowed.
- An exported generic entity uses an unexported declaration as a type argument, type parameter constraint or a type parameter default (including cases when the type argument is not used elsewhere in exported the declaration);
- An unexported class or interface is used in `extends` clause of exported class or interface, or an unexported interface is used in `implements` clause of an exported class. A public or protected class field uses an unexported entity as a type. A public or protected method of exported class uses an unexported entity in its signature;
- An exported type alias declaration uses unexported type;
- An annotation which is applied to the exported declaration is not exported or uses an unexported type;
- An exported overload contains one or more unexported entities;

Here is the series of examples representing that cases:

1. An exported declarations without explicit types or explicit return types.

```
// // functions, variables, constants
export let a: int = 1 // OK
export let b = 1 // Compile-time error - no explicit type

export const c: int = 1 // OK
export const d = 1 // Compile-time error - no explicit type

export function foo(): void {} // OK

export function bar() {}  // Compile-time error, no return type

// // Fields and methods
export class C {
   // static and instance fields and methods (public and protected)
   x = 1   // Compile-time error, no explicit type for a public field
   protected xx = 1   // Compile-time error, no explicit type for a protected field
   static v = 1 // Compile-time error, no explicit type for public field
   y: int = 1   // OK
   private z = 1 // OK - not a public/protected field

   foo() { return 1 } // Compile-time error, no explicit return type
                      // for a public method

   static bar() {} // Compile-time error, no explicit
                   // return type for a public static method
   static bar_ok(): void {} // OK


   protected i_bar() { return 1 } // Compile-time error, no return
                                  // type for a protected method
```

(continues on next page)

(continued from previous page)

```
   protected bar_ok(): int { return 1 } // OK

   private baz() { return "hello" } // OK, baz() is private
}

// // Interfaces
// OK
export interface I {
   get_count(): number { return 1; }
   set_count(n: number): void {}
}

// Compile-time error, no explicit return types for get_count and set_count
export interface J {
   get_count() { return 1; }
   set_count(n: number) {}
}

// // getters
let _counter: int = 1  // OK
let _name = "empty"          // OK
export get counter() { return _counter } // Compile-time error, no return type
export get name(): string { return _name } // OK

export namespace NS {
   get name() { return "Bob" } // OK since not exported
   export get age(): int { return 1 }  // OK
   export get sex() { return "male" } // Compile-time error, no explicit
                                      // return type
}
```

2. An exported function, a variable, a constant, or an accessor with unexported entity.

```
class A { constructor(a: A) {}; };

// The following declarations cause compile-time errors
// because 'A' is not exported
export let v: A
export function foo (p: A): A { return p; }
export const x3: A = new A()
export get val(): A { return new A() }
export set val(a: A) {}

export class B { constructor(B: B) {}; };
class C extends B {};

// Next is OK, can use an unexported default as an optional parameter
export function bar(p: B = new C() ) {}
```

3. An exported generic uses an unexported entity.

```
class Arg {};

// OK since T is a type parameter
export class G<T> {}

// Compile-time error, type parameter default 'Arg' not exported
export function foo<T = Arg>(): void {}
// Compile-time error, type parameter constraint 'Arg' not exported
export function foo<T extends Arg>(): void {}
```

4. An exported class or interface uses an unexported entity as a type of a public or protected field or in a signature of public or protected method.

```
class C { constructor() {} ; };
interface I {};

// // unexported type in extends/implements
// Compile-time error, 'C' and 'I' must be exported
export class C1 extends C implements I {}
// Compile-time error, 'I' must be exported
export interface I1 extends I {}

// // unexported entity inside a declaration
export class C1 {
   // // Compile-time errors due to unexported 'C'
   f: C = new C;
   doIt(): C { return new C(); }
   protected tryMe(): C { return new C(); }

   // // OK, unexported 'C' can be used in private members/fields
   private secret(): C { return new C(); }
}
```

5. An exported type alias declaration refers to an unexported type.

```
 class C {};

 // Compile-time error, 'C' must be exported
 export type A = C
```

6. An annotation applied to an exported declaration not exported or uses an unexported type.

```
type Version = number[];

// compile time error, ``Version`` not exported
export @interface deprecated {
```

(continues on next page)

(continued from previous page)

```
                    fromVersion: Version;
                }

export @deprecated([1, 1]) function bar() {};

|
```

7. One or more unexported entities in an exported overload:

```
function foo(): void  {};
export function bar(): void  {};

// Compile-time error, `foo` not exported
export overload baz { foo, bar }
```

13.7 Export Directives

Export directive allows the following:

- Specifying a selective list of exported declarations with optional renaming;
- Specifying a name of one declaration;
- Exporting a type; or
- Re-exporting declarations from other modules.

The syntax of an *export directive* is presented below:

```
exportDirective:
    selectiveExportDirective
    | singleExportDirective
    | exportTypeDirective
    | reExportDirective
    ;
```

Limitations on exported declarations are described with examples in *Exported Declarations*.

13.7.1 Selective Export Directive

Top-level declarations can be made *exported* by using a selective export directive. The selective export directive provides an explicit list of names of the declarations to be exported. Optional renaming allows having the declarations exported with new names.

The syntax of *selective export directive* is presented below:

```
selectiveExportDirective:
    'export' selectiveBindings
    ;
```

A selective export directive uses the same *selective bindings* as an import directive:

```
export { d1, d2 as d3}
```

The above directive exports 'd1' by its name, and 'd2' as 'd3'. The name 'd2' is not accessible (see *Accessible*) in the modules that import this module.

13.7.2 Single Export Directive

Single export directive allows specifying the declaration to be exported from the current module by using the declaration's own name, or anonymously.

The syntax of *single export directive* is presented below:

```
singleExportDirective:
    'export'
    ( 'type'? identifier
    | 'default' (expression | identifier)
    | '{' identifier 'as' 'default' '}'
    )
    ;
```

If `default` is present, then only one such export directive is possible in the current module. Otherwise, a compile-time error occurs.

The directive in the example below exports variable 'v' by its name:

```
export v
let v: int = 1
```

The directive in the example below exports class 'A' by its name as default export:

```
class A {}
export default A
export {A as default} // such syntax is also acceptable
```

The directive in the example below exports a constant variable anonymously:

```
class A {}
export default new A
```

Single export directive acts as re-export when the declaration referred to by *identifier* is imported.

```
import {v} from "some location"
export v
```

13.7.3 Export Type Directive

An export directive can have a `type` modifier exclusively for a better syntactic compatibility with TypeScript (also see *Import Type Directive*).

The *export type directive* syntax is presented below:

```
exportTypeDirective:
    'export' 'type' selectiveBindings
    ;
```

ArkTS supports no additional semantic checks for entities exported by using *export type* directives.

If a binding refers to something other than `type`, then a compile-time error occurs.

13.7.4 Re-Export Directive

In addition to exporting what is declared in the module, it is possible to re-export declarations that are part of other modules' export. A particular declaration or all declarations can be re-exported from a module. When re-exporting, new names can be given. This action is similar to importing but has the opposite direction.

The syntax of *re-export directive* is presented below:

```
reExportDirective:
    'export'
    ('*' bindingAlias?
    | selectiveBindings
    | '{' 'default' bindingAlias? '}'
    )
    'from' importPath
    ;
```

An `importPath` cannot refer to the file the current module is stored in. Otherwise, a compile-time error occurs.

If re-exported declarations are not distinguishable (see *Declarations*) within the scope of the current module, then a compile-time error occurs.

The re-exporting practices are represented in the following examples:

```
export * from "path_to_the_module" // re-export all exported declarations
export * as qualifier from "path_to_the_module"
   // re-export all exported declarations with qualification
export { d1, d2 as d3} from "path_to_the_module"
   // re-export particular declarations some under new name
```

(continues on next page)

(continued from previous page)

```
export {default} from "path_to_the_module"
   // re-export default declaration from the other module
export {default as name} from "path_to_the_module"
   // re-export default declaration from the other module under 'name'
```

13.8 Top-Level Statements

A module can contain sequences of statements that logically comprise one sequence of statements.

The syntax of *top-level statements* is presented below:

```
topLevelStatements:
    statement*
    ;
```

A module can contain any number of top-level statements that logically merge into a single sequence in the textual order:

```
   statements_1
   /* top-declarations except constant and variable declarations */
   statements_2
```

The sequence above is equal to the following:

```
   /* top-declarations except constant and variable declarations */
   statements_1; statements_2
```

This situation is represented by the example below:

```
// The actual text combination of the statements and declarations
console.log ("Start of top-level statements")
type A = number | string
let a: A = 56
function foo () {
   console.log (a)
}
a = "a string"


// The logically ordered text - declarations then statements
type A = number | string
function foo () {
   console.log (a)
}
console.log ("Start of top-level statements")
let a: A = 56
a = "a string"
```

- If a module is imported by some other module, then the semantics of top-level statements is to initialize the imported module. It means that all top-level statements are executed only once before a call to any other function, or before the access to any top-level variable of the module.
- If a module is used as a program, then top-level statements are used as a program entry point (see *Program Entry Point*). The set of top-level statements being empty implies that the program entry point is also empty and does nothing. If a module has the `main` function, then it is executed after the execution of the top-level statements.

```
// Source file A
{ // Block form
  console.log ("A.top-level statements")
}

// Source file B
import * as A from "Source file A "
function main () {
   console.log ("B.main")
}
```

The output is as follows:

A. Top-level statements,

B. Main.

```
// One source file
console.log ("A.Top-level statements")
function main () {
   console.log ("B.main")
}
```

A compile-time error occurs if top-level statements contain a return statement (*Expression Statements*).

The execution of top-level statements means that all statements, except type declarations, are executed one after another in the textual order of their appearance within the module until an error is thrown (see *Errors*), or last statement is executed.

Thus, if a top-level statement refers to a variable or constant and the declaration of that variable or constant (see *Variable and Constant Declarations*) is textually located after the the current statement, then a compile-time error occurs.

```
console.log (a, b) // Compile-time error
let a = 1
const b = a
```

See details for variable or constant declarations validity in *Validity of Initializer*.

13.9 Multifile Module

Multifile module is a module that consists of several source files which have the same *module header* (see *Module Header*).

If two *module headers* (see *Module Header*) have the same *moduleName* but different `export` modifiers, then a compile-time error occurs.

A *multifile module* combines *Import Directives*, *Top-Level Declarations*, and *Export Directives* for all files of the module.

A *multifile module* has the following limitations:

- If top-level statements (see *Top-Level Statements*) are located in different files, then a compile-time error occurs.

A correct *multifile module* is represented in the example below:

```
// file1
export module "x"
import {A} from "some module"
export a

// file2
export module "mod1"
let a = new A()
```

```
// file1
module "x"
function foo() {}
function bar() {}
namespace NS1 {
    function foo() {}
    function bar() {}
}

// file2
module "x"
class A {}
```

An incorrect *multifile module* is represented in the example below:

```
// file1
module "y"
let a = 8
namespace NS1 {
    let a = 9
}

// file2
module "y"
let b = 4       // Compile-time error, the top-level statements located in several files
namespace NS1 {
    let b = 6   // Compile-time error, the top-level statements located in several files
}
```

13.10 Standard Library Usage

A set of entities exported from the standard library (see *Standard Library*) is accessible as simple names (see *Accessible*) at module scope and in nested scopes if not redefined. Using these names as names of programmer-defined entities at the module scope causes a compile-time error as discussed in *Declarations*.

```
console.log("Hello, world!") // OK, 'console' is defined in the standard library

let console = 5 // Compile-time error
```

13.11 Program Entry Point

Modules can act as programs (applications). Program execution starts from the execution of a *program entry point* which can be of the following two kinds:

- Top-level statements for modules (see *Top-Level Statements*); or
- Entry point function (see below).

A module can have the following forms of entry point:

- Sole entry point function (`main` or other as described below);
- Sole top-level statement (the first statement in the top-level statements acts as the entry point);
- Both top-level statement and entry point function (same as above, plus the function called after the top-level statement execution is completed).

Entry point functions have the following features:

- Any exported top-level function can be used as an entry point. An entry point is selected by the compiler, the execution environment, or both;
- Entry point function must either have no parameters, or have one parameter of type `FixedArray<string>` that provides access to the arguments of a program command line;
- Entry point function return type is either `void` (see *Type void or undefined*) or `int`;
- Entry point function cannot be overloaded;
- Entry point function is called `main` by default.

Different forms of valid and invalid entry points are represented in the example below:

```
function main() {
  // Option 1: a return type is inferred from the body of main().
  // It will be 'int' if the body has 'return' with the integer expression
  // and 'void' if no return at all in the body
}

function main(): void {
  // Option 2: explicit :void - no return in the function body required
}

function main(): int {
```

(continues on next page)

(continued from previous page)

```
  // Option 3: explicit :int - return is required
  return 0
}

function main(): string { // Compile-time error, incorrect main signature
  return ""
}

function main(p: number) { // Compile-time error, incorrect main signature
}

// Option 4: top-level statement is the entry point
console.log ("Hello, world!")

// Option 5: top-level exported function
export function entry(): void {}

// Option 5: top-level exported function with command-line arguments
export function entry(cmdLine: FixedArray<string>): void {}
```

CHAPTER

FOURTEEN

AMBIENT DECLARATIONS

Ambient declaration specifies an entity declared elsewhere but usable in the current context.

Ambient declarations:

- Provide type information for entities declared elsewhere.
- Introduce no new entities like regular declarations do.
- Cannot include executable code, and thus
  - Ambient variables, constants, and enumerations have no initializers;
  - Ambient functions, methods, and constructors have no bodies.

The syntax of *ambient declaration* is presented below:

```
ambientDeclaration:
    'declare'
    ( ambientConstantOrVariableDeclaration
    | ambientFunctionDeclaration
    | explicitFunctionOverload
    | ambientClassDeclaration
    | ambientInterfaceDeclaration
    | ambientEnumDeclaration
    | ambientNamespaceDeclaration
    | ambientAnnotationDeclaration
    | ambientAccessorDeclaration
    | typeAlias
    )
    ;
```

A compile-time error occurs if the modifier `declare` is used in a context that is already ambient:

```
declare namespace A{
    declare function foo(): void // Compile-time error
}
```

Ambient declaration by itself does not guarantee that non-ambient declartion declared elsewhere for the same name entity is identical to the ambient one. By its nature, any *ambient declaration* is a requirement only that the build system (see *Build System*) provides a proper non-ambient declaration.

14.1 Ambient Constant or Variable Declarations

The syntax of *ambient* constant or variable declarations is presented below:

```
ambientConstantOrVariableDeclaration:
    'const'|'let' ambientConstantOrVariableList ';'
    ;

ambientConstantOrVariableList:
    ambientConstantOrVariable (',' ambientConstantOrVariable)*
    ;

ambientConstantOrVariable:
    identifier ':' type
    ;
```

An *ambient constant* and *variable declaration* must have an explicit type annotation, and must have no initializer. Otherwise, a compile-time error occurs:

```
declare let v1: number // OK
declare let v2 = 1     // Compile-time error, ambient variable must have no initializer

declare const c1: number // OK
declare const c2 = 1     // Compile-time error, ambient constant must have no␣
↪initializer
```

14.2 Ambient Function Declarations

The syntax of *ambient function declaration* is presented below:

```
ambientFunctionDeclaration:
    'function' identifier typeParameters? signature
    ;
```

A compile-time error occurs if an ambient function declaration has the following:

- No explicit return type specified;
- A parameter with the default value;
- A function body, or;
- Modifier `async` specified.

Examples below illustrate that:

```
declare function ok1(x: number): void // OK
declare function bad1(x: number) // Compile-time error, no return specified

declare function ok2(x?: string): void // OK, optional parameters can be used
declare function bad2(y: number = 1): void // Compile-time error, parameter
```

(continues on next page)

(continued from previous page)

```
                                        // has default value

declare function bad3(): void {} // Compile-time error, function body provided

declare async function bad4(): void // Compile-time error, async modifier is used
```

14.3 Ambient Overload Function Declarations

The syntax of *ambient overload function declaration* is identical to that of *Explicit Function Overload*. The semantics of such declarations is defined by the same rules.

```
// Top-level functions are overloaded
declare function foo1(p: string): void
declare function foo2(p: number): void
declare overload foo {foo1, foo2}

// Namespace functions are overloaded
declare namespace N {
   function foo1(p: string): void
   function foo2(p: number): void
   overload foo {foo1, foo2}
}

// All calls are valid
foo("a string")
foo(5)
N.foo("a string")
N.foo(5)
```

14.4 Ambient Class Declarations

The syntax of *ambient class declaration* is presented below:

```
ambientClassDeclaration:
    'class'|'struct' identifier typeParameters?
    classExtendsClause? implementsClause?
    '{' ambientClassMember* '}'
    ;

ambientClassMember:
    ambientAccessModifier?
    ( ambientFieldDeclaration
```

(continues on next page)

(continued from previous page)

```
    | ambientConstructorDeclaration
    | ambientMethodDeclaration
    | explicitClassMethodOverload
    | ambientClassAccessorDeclaration
    | ambientIndexerDeclaration
    | ambientCallSignatureDeclaration
    | ambientIterableDeclaration
    )
    ;

ambientAccessModifier:
    'public' | 'protected'
    ;
```

Ambient field declarations have no initializers.

The syntax of *ambient field declaration* is presented below:

```
ambientFieldDeclaration:
    ambientFieldModifier* identifier ':' type
    ;

ambientFieldModifier:
    'static' | 'readonly'
    ;
```

Ambient constructor, method, and accessor declarations have no bodies.

Their syntax is presented below:

```
ambientConstructorDeclaration:
    'constructor' parameters
    ;

ambientMethodDeclaration:
    ambientMethodModifier* identifier signature
    ;

ambientMethodModifier:
    'static'
    ;

ambientClassAccessorDeclaration:
    ambientMethodModifier*
    ( 'get' identifier '(' ')' returnType
    | 'set' identifier '(' requiredParameter ')'
    )
    ;
```

Ambient methods can be overloaded similarly to non-ambient methods with the same syntax and semantics (see *Explicit Class Method Overload*).

```
1 // Class methods are overloaded
2 declare class A {
```

(continues on next page)

(continued from previous page)

```
   foo1(p: string): void
   foo2(p: number): void
   overload foo {foo1, foo2}
}

// All methods calls are valid
function demo (a: A) {
   a.foo("a string")
   a.foo(5)
}
```

14.4.1 Ambient Indexer

Ambient indexer declarations specify the indexing of a class instance in an ambient context. The feature is provided for TypeScript compatibility:

The syntax of *ambient indexer declaration* is presented below:

```
ambientIndexerDeclaration:
    'readonly'? '[' identifier ':' type ']' returnType
    ;
```

The use of *ambient indexer declarations* is represented in the example below:

```
declare class C {
    [index: number]: number
}
declare class D {
    [index: int]: C
}
declare class E {
    [index: string]: string
}
```

The following restrictions apply:

- Only one *ambient indexer declaration* is allowed in an ambient class declaration.
- *Ambient indexer declaration* is supported in ambient contexts only. If written in ArkTS, ambient class implementation must conform to *Indexable Types*.

14.4.2 Ambient Call Signature

Ambient call signature declarations are used to specify *callable types* in an ambient context. The feature is provided for TypeScript compatibility:

The syntax of *ambient call signature declaration* is presented below:

```
ambientCallSignatureDeclaration:
    signature
    ;
```

```
declare class C {
    (someArg: number): boolean
    (someArg: string): boolean
    ...
}
```

Ambient class signature declaration is supported in ambient contexts only. If written in ArkTS, ambient class implementation must conform to *Callable Types with $_invoke Method*.

Multiple *ambient call signatures* are allowed in an ambient class declaration provided that they are distinct (see *Declaration Distinguishable by Signatures*). Multiple distinct ambient call signatures are represented in the following example:

```
// sdk_file.d.ets, declaration file
export declare class C {
   (x: string): void
   (x: number): void
}

// sdk_file.ets, implementation file
export class C {
   static $_invoke(x: string): void {
      console.log('string')
   }
   static $_invoke(x: number): void {
      console.log('number')
   }
}

// app.ets
import { C } from './sdk_file'

C(123)    // log: number
C('abc')  // log: string
```

14.4.3 Ambient Iterable

Ambient iterable declaration indicates that a class instance is iterable in an ambient context. The feature is provided for TypeScript compatibility:

The syntax of *ambient iterable declaration* is presented below:

```
ambientIterableDeclaration:
    '[Symbol.iterator]' '(' ')' returnType
    ;
```

The following restrictions apply:

- *returnType* must be a type that implements `Iterator` interface defined in *Standard Library*.
- Only one *ambient iterable declaration* is allowed in an ambient class declaration.

```
declare class C {
    [Symbol.iterator] (): CIterator
}
```

Note

Ambient iterable declaration is supported in ambient contexts only. If written in ArkTS, ambient class implementation must conform to *Iterable Types*.

14.5 Ambient Interface Declarations

The syntax of *ambient interface declaration* is presented below:

```
ambientInterfaceDeclaration:
    'interface' identifier typeParameters?
    interfaceExtendsClause?
    '{' ambientInterfaceMember* '}'
    ;

ambientInterfaceMember
    : interfaceProperty
    | ambientInterfaceMethodDeclaration
    | ambientIndexerDeclaration
    | ambientIterableDeclaration
    ;

ambientInterfaceMethodDeclaration:
    'default'? identifier signature
    ;
```

Ambient interface can contain additional members in the same manner as an ambient class (see *Ambient Indexer*, and *Ambient Iterable*).

If an interface method declaration is marked with the keyword `default`, then a non-ambient interface must contain the default implementation for the method as follows:

```
declare interface I1 {
    default foo (): void // method foo will have the default implementation
}
class C1 implements I1 {} // Class C1 is valid as foo() has the default implementation

interface I1 {
    // If such interface is used as I1 it will be runtime error as there is
```

(continues on next page)

(continued from previous page)

```
    // no default implementation for foo()
    foo (): void
}

declare interface I2 {
    foo (): void // method foo has no default implementation
}
class C2 implements I2 {} // Class C2 is invalid as foo() has no implementation
class C3 implements I2 { foo() {} } // Class C3 is valid as foo() has implementation
```

14.6 Ambient Enumeration Declarations

The syntax of *ambient enumeration declaration* is presented below:

```
ambientEnumDeclaration
    : 'const'? 'enum' identifier enumBaseType? '{' ambientEnumMemberList? '}'
    ;

ambientEnumMemberList:
    identifier (',' identifier)* ','?
    ;
```

If an *enumeration declaration* is prefixed with the keyword `const`, then a compile-time error occurs. This restriction is temporary, and the semantics of `const enum` is to be made available in the future versions of ArkTS.

No member of an enum declaration can have an initializer. Otherwise, a compile-time error occurs as represented in the example below:

```
declare enum RGB {Red, Green, Blue} // OK

declare enum Err1 { A = 5 }      // Compile-time error, initializer is present
```

14.7 Ambient Namespace Declarations

Namespaces are used to logically group multiple entities. ArkTS supports *ambient namespaces* for better TypeScript compatibility. TypeScript often uses ambient namespaces to specify the platform API or a third-party library API.

The syntax of *ambient namespace declaration* is presented below:

```
ambientNamespaceDeclaration:
    'namespace' identifier '{' ambientNamespaceElement* '}'
    ;

ambientNamespaceElement:
```

(continues on next page)

(continued from previous page)

```
    ambientNamespaceElementDeclaration | exportDirective
;

ambientNamespaceElementDeclaration:
    'export'?
    ( ambientConstantOrVariableDeclaration
    | ambientFunctionDeclaration
    | ambientClassDeclaration
    | ambientInterfaceDeclaration
    | ambientNamespaceDeclaration
    | ambientAccessorDeclaration
    | 'const'? enumDeclaration
    | typeAlias
    )
    ;
```

An *enumeration type declaration* can be prefixed with the keyword `const` for TypeScript compatibility. The prefix has no influence on the declared type. Only exported entities can be accessed outside a namespace.

Namespaces can be nested:

```
declare namespace A {
    export namespace B {
        export function foo(): void;
    }
}
```

A namespace is not an object but merely a scope for entities that can be accessed by using qualified names only.

If an ambient namespace is imported from a module, then all ambient namespace declarations are accessible (see *Accessible*) across all declarations and top-level statements of the current module.

```
// File1.d.ets
export declare namespace A { // namespace itself must be exported
    function foo(): void
    type X = Array<number>
}

// File2.ets
import {A} from 'File1.d.ets'

A.foo() // Valid function call, as 'foo' is accessible for top-level statements
function foo () {
    A.foo() // Valid function call, as 'foo' is accessible here as well
}
class C {
    method () {
        A.foo() // Valid function call, as 'foo' is accessible here too
        let x: A.X = [] // Type A.X can be used
    }
}
```

A compile-time error occurs if an *ambient namespace* declaration contains an *exportDirective* that refers to a declaration which is not a part of the namespace.

```
export declare namespace A {
     export {foo} // Compile-time error, no 'foo' in namespace 'A'
}
function foo() {}
```

14.7.1 Implementing Ambient Namespace Declaration

If an *ambient namespace* is implemented in ArkTS, a namespace with the same name must be declared (see *Namespace Declarations*) as the top-level declaration of a module. All namespace names of a nested namespace (i.e. a namespace embedded into another namespace) must be the same as in ambient context.

14.8 Ambient Accessor Declarations

Ambient accessor declaration is an ambient version of *Accessor Declarations*. The syntax of an *ambient accessor declaration* is presented below:

```
ambientAccessorDeclaration:
    ( 'get' identifier '(' receiverParameter? ')' returnType
    | 'set' identifier '(' (receiverParameter ',')? requiredParameter ')'
    )
    ;
```

A compile-time error occurs if explicit return type for an ambient getter declaration is not specified.

```
declare get name(): string // OK
declare get age() // Compile-time error, return type must be specified
```

See *Accessor Declarations* for details.

CHAPTER

FIFTEEN

SEMANTIC RULES

This Chapter contains semantic rules to be used throughout this Specification document. The description of the rules is more or less informal. Some details are omitted to simplify the understanding.

15.1 Semantic Essentials

The section gives a brief introduction to the major semantic terms and their usage in several contexts.

15.1.1 Type of Standalone Expression

Standalone expression (see *Type of Expression*) is an expression for which there is no target type in the context where the expression is used.

The type of a *standalone expression* is determined as follows:

- In case of *Numeric Literals*, the type is the default type of a literal:
  - Type of *Integer Literals* is `int` or `long`;
  - Type of *Floating-Point Literals* is `double` or `float`.
- In case of *Constant Expressions*, the type is inferred from operand types and operations.
- In case of an *Array Literal*, the type is inferred from the elements (see *Array Type Inference from Types of Elements*).
- Otherwise, a compile-time error occurs. Specifically, a compile-time error occurs if an *object literal* is used as a *standalone expression*.

The situation is represented in the example below:

```
function foo() {
  1    // type is 'int'
  1.0  // type is 'number'
  [1.0, 2.0]  // type is number[]
  [1, "aa"] // type is (int | string)
}
```

15.1.2 Specifics of Assignment-like Contexts

Assignment-like context (see *Assignment-like Contexts*) can be considered as an assignment `x = expr`, where `x` is a left-hand-side expression, and `expr` is a right-hand-side expression. E.g., there is an implicit assignment of `expr` to the formal parameter `foo` in the call `foo(expr)`, and implicit assignments to elements or properties in *Array Literal* and *Object Literal*.

Assignment-like context is specific in that the type of a left-hand-side expression is known, but the type of a right-hand-side expression is not necessarily known in the context as follows:

- If the type of a right-hand-side expression is known from the expression itself, then the *Assignability* check is performed as in the example below:

```
function foo(x: string, y: string) {
    x = y // OK, assignability is checked
}
```

- Otherwise, an attempt is made to apply the type of the left-hand-side expression to the right-hand-side expression. A compile-time error occurs if the attempt fails as in the example below:

```
function foo(x: int, y: double[]) {
    x = 1 // OK, type of '1' is inferred from type of 'x'
    y = [1, 2] // OK, array literal is evaluated as [1.0, 2.0]
}
```

15.1.3 Specifics of Variable Initialization Context

If the variable or a constant declaration (see *Variable and Constant Declarations*) has an explicit type annotation, then the same rules as for *assignment-like contexts* apply. Otherwise, there are two cases for `let x = expr` (see *Type Inference from Initializer*) as follows:

- The type of the right-hand-side expression is known from the expression itself, then this type becomes the type of the variable as in the example below:

```
function foo(x: int) {
    let y = x // type of 'y' is 'int'
}
```

- Otherwise, the type of `expr` is evaluated as type of a standalone expression as in the example below:

```
function foo() {
    let x = 1 // x is of type 'int' (default type of '1')
    let y = [1, 2] // x is of type 'number[]'
}
```

15.1.4 Specifics of Numeric Operator Contexts

The `postfix` and `prefix` `increment` and `decrement` operators evaluate `byte` and `short` operands without widening. It is also true for an `assignment` operator (considering `assignment` as a binary operator).

For other numeric operators, the operands of unary and binary numeric expressions are widened to a larger numeric type. The minimum type is `int`. None of those operators evaluates values of types `byte` and `short` without widening. Details of specific operators are discussed in corresponding sections of the Specification.

15.1.5 Specifics of String Operator Contexts

If one operand of the binary operator `'+'` is of type `string`, then the string conversion applies to another non-string operand to convert it to string (see *String Concatenation* and *String Operator Contexts*).

15.1.6 Other Contexts

The only semantic rule for all other contexts, and specifically for *Overriding*, is to use *Subtyping*.

15.1.7 Specifics of Type Parameters

If the type of a left-hand-side expression in *assignment-like context* is a type parameter, then it provides no additional information for type inference even where a type parameter constraint is set.

If the *target type* of an expression is a *type parameter*, then the type of the expression is inferred as the type of a *standalone expression*.

The semantics is represented in the example below:

```
class C<T extends number> {
    constructor (x: T) {}
}

new C(1) // Compile-time error
```

The type of `'1'` in the example above is inferred as `int` (default type of an integer literal). The expression is considered `new C<int>(1)` and causes a compile-time error because `int` is not a subtype of `number` (type parameter constraint).

Explicit type argument `new C<number>(1)` must be used to fix the code.

15.1.8 Semantic Essentials Summary

Major semantic terms are listed below:

- *Type of Expression*;
- *Assignment-like Contexts*;
- *Type Inference from Initializer*;
- *Numeric Operator Contexts*;
- *String Operator Contexts*;
- *Subtyping*;
- *Assignability*;
- *Overriding*;
- *Overloading*;
- *Type Inference*.

15.2 Subtyping

Subtype relationship between types `S` and `T`, where `S` is a subtype of `T` (recorded as `S<:T`), means that any object of type `S` can be safely used in any context to replace an object of type `T`. The opposite relation (recorded as `T:>S`) is called *supertype* relationship. Each type is its own subtype and supertype (`S<:S` and `S:>S`).

By the definition of `S<:T`, type `T` belongs to the set of *supertypes* of type `S`. The set of *supertypes* includes all *direct supertypes* (discussed in subsections), and all their respective *supertypes*. More formally speaking, the set is obtained by reflexive and transitive closure over the direct supertype relation.

The terms *subclass*, *subinterface*, *superclass*, and *superinterface* are used in the following sections as synonyms for *subtype* and *supertype* when considering non-generic classes, generic classes, or interface types.

If a relationship of two types is not described in one of the following sections, then the types are not related to each other. Specifically, two *Resizable Array Types* and two *Tuple Types* are not related to each other, except where they are identical (see *Type Identity*).

```
class Base {}
class Derived extends Base {}

function not_a_subtype (
   ab: Array<Base>, ad: Array<Derived>,
   tb: [Base, Base], td: [Derived, Derived],
) {
   ab = ad // Compile-time error
   tb = td // Compile-time error
}
```

15.2.1 Subtyping for Non-Generic Classes and Interfaces

S for non-generic classes and interfaces is a direct *subclass* or *subinterface* of T (or of Object type) when one of the following conditions is true:

- Class S is a *direct subtype* of class T (S<:T) if T is mentioned in the extends clause of S (see *Class Extension Clause*):

```
// Illustrating S<:T
class T {}
class S extends T {}
function foo(t: T) {}

// Using T
foo(new T)

// Using S (S<:T)
foo(new S)
```

- Class S is a *direct subtype* of class Object (S<:Object) if S has no *Class Extension Clause*:

```
// Illustrating S<:Object
class S {}
function foo(o: Object) {}

// Using Object
foo(new Object)

// Using S (S<:Object)
foo(new S)
```

- Class S is a *direct subtype* of interface T (S<:T) if T is mentioned in the implements clause of S (see *Class Implementation Clause*):

```
// Illustrating S<:T
// S is class, T is interface
interface T {}
class S implements T {}
function foo(t: T) {}
let s: S = new S

// Using T
let t: T = s
foo(t)

// Using S (S<:T)
foo(s)
```

- Interface S is a *direct subtype* of interface T (S<:T) if T is mentioned in the extends clause of S (see *Superinterfaces and Subinterfaces*):

```
// Illustrating S<:T
// S is interface, T is interface
interface T {}
interface S extends T {}
```

(continues on next page)

(continued from previous page)

```
function foo(t: T) {}
let t: T
let s: S

// Using T
class A implements T {}
t = new A
foo(t)

// Using S (S<:T)
class B implements S {}
s = new B
foo(s)
```

- Interface `S` is a *direct subtype* of class `Object` (`S<:Object`) if `S` has no `extends` clause (see *Superinterfaces and Subinterfaces*).

```
// Illustrating subinterface of Object
interface S {}
function foo(o: Object) {}

// Using Object
foo(new Object)

// Using subinterface of Object
class A implements S {}
let s: S = new A;
foo(s)
```

15.2.2 Subtyping for Generic Classes and Interfaces

A *generic class* or *generic interface* is declared as `C <`F_1 `, ..., `F_n`>`, where $n>0$ is a *direct subtype* of another generic class or interface `T`, if one of the following conditions is true:

- `T` is a *direct superclass* of `C <`F_1 `, ..., `F_n`>` mentioned in the `extends` clause of `C`:

```
// T<U, V>  is direct superclass of C<U,V>
// T<U, V> >: C<U, V>

class T<U, V> {
   foo(p: U|V): U|V { return p }
}

class C<U, V> extends T<U, V> {
   bar(u: U): U { return u }
}


// OK, exact match
```

(continues on next page)

(continued from previous page)

```
let t: T<int, boolean>  = new T<int, boolean>
let c: C<int, boolean> = new C<int, boolean>


// OK, assigning to direct superclass
t =  new C<int, boolean>

// Compile-time error, cannot assign to subclass
c =  new T<int, boolean>
```

- T is one of direct superinterfaces of C <F_1 , . . . , F_n> (see *Superinterfaces and Subinterfaces*):

```
// Interface I<U, V>  is direct superinterface
// of J<U,V>, X<U, V>

interface I<U, V> {
   foo(u: U): U;
   bar(v: V): V;
}

// J<U, V> <: I<U, V>
// since J extends I
interface J<U, V> extends I<U, V>
{
   foo(u: U): U
   bar(v: V): V

   foo1(p: U|V): U|V
}

// X<U, V> <: I<U, V>
// since X implements I
class X<U, V> implements I<U,V> {
  foo(u: U): U { return u }
  bar(v: V): V { return v }
}

// Y<U,V> <: J<U, V>  (directly)
// Also Y<U, V> <: I<U, V> (transitively)
class Y<U, V> implements J<U,V> {
  foo(u: U): U { return u }
  bar(v: V): V { return v }

  foo1(p: U|V): U|V { return p }
}

let i: I<int, boolean>
let j: J<int, boolean>
let x = new X<int, boolean>
let y = new Y<int, boolean>

// OK, assigning to direct supertypes
```

(continues on next page)

(continued from previous page)

```
i = x
j = y

// OK, assigning subinterface (J<:I)
i = j

// Compile-time error, cannot assign superinterface (I>:JJ
j = i
```

- T is type `Object` (C<:Object) if `C <`F_1 `, . . . ,` F_n`>` is either a generic class type with no *direct superclasses*, or a generic interface type with no direct superinterfaces:

```
// Object is direct superclass and for C<U,V>
// and direct superinterface for I<U,V>
//
class C<U, V> {
   foo(u: U): U { return u }
   bar(v: V): V { return v }
}
interface I<U, V> {
   foo(u: U): U { return u }
   bar(v: V): V { return v }
}

let o: Object = new Object
let c: C<int, boolean> = new C<int, boolean>
let i: I<int, boolean>

// // example1 - C<U,V> <: Object
function example1(o: Object) {}

// OK, example(Object)
example1(o)
// OK, C<int, boolean> <: Object
example1(c)

// // example2 - I<U,V> <: Object
function example2(o: Object) {}
class D<U, V> implements I<U, V> {}
i = new D<int, boolean>

// OK, example2(Object)
example2(o)
// OK, I<int, boolean> <: Object
example2(i)
```

The direct supertype of a type parameter is the type specified as the constraint of that type parameter.

If type parameters of a generic class or an interface have a variance specified (see *Type Parameter Variance*), then the subtyping for instantiations of the class or interface is determined in accordance with the variance of the appropriate type parameter. For example, if a generic class `G<in T1,out T2>` is defined, `S>:U` and `T<:V`, then `G<S,T> <: G<U, V>`. It is represented by the following code:

```
// Subtyping illustration for a generic with parameter variance

// U1 <: U0
class U0 {}
class U1 extends U0 {}

// Generic with contravariant parameter
class E<in T> {}

let e0: E<U0> = new E<U1> // Compile-time error, E<U0> is subtype of E<U1>
let e1: E<U1> = new E<U0> // OK, E<U1> is supertype for E<U0>

// Generic with covariant parameter
class F<out T> {}

let f0: F<U0> = new F<U1> // OK, F<U0> is supertype for F<U1>
let f1: F<U1> = new F<U0> // Compile-time error, F<U1> is subtype of F<U0>
```

15.2.3 Subtyping for Literal Types

Any `string` literal type (see *String Literal Types*) is *subtype* of type `string`. It affects overriding as shown in the example below:

```
class Base {
    foo(p: "1"): string { return "42" }
}
class Derived extends Base {
    override foo(p: string): "1" { return "1" }
}
// Type "1" <: string

let base: Base = new Derived
let result: string = base.foo("1")
/* Argument "1" (value) is compatible to type "1" and to type string in
   the overridden method
   Function result of type string accepts "1" (value) of literal type "1"
 */
```

Literal type `null` (see *Literal Types*) is a subtype and a supertype to itself. Similarly, literal type `undefined` is a subtype and a supertype to itself.

15.2.4 Subtyping for Tuple Types

Any tuple type is a subtype of type `Tuple` (see *Type Tuple*).

Tuple type `T` (P_1 , ... , P_n) is a subtype of type `S` (P_1 , ... , P_m), where $n \geq m$. I.e., a tuple type is a subtype of a tuple type with fewer identical constituent types (*Type Identity*).

```
function foo(t1: [number], t2: [string, number]) {
    let a: [] = t1       // OK
    let b: [string] = t2 // OK

    t1 = t2 // Compile-time error
    t2 = t1 // Compile-time error

    let d: [string, number, boolean] = ["a", 1, true]
    let t2 = d // OK
    let d = t2 // Compile-time error
}
```

15.2.5 Subtyping for Union Types

A union type `U` participates in a subtyping relationship (see *Subtyping*) in the following cases:

1. Union type `U` (U_1 | ... | U_n) is a subtype of type `T` if each U_i is a subtype of `T`.

```
let s1: "1" | "2" = "1"
let s2: string = s1 // OK

let a: string | number | boolean = "abc"
let b: string | number = 42
a = b // OK
b = a // Compile-time error, boolean is absent is 'b'

class Base {}
class Derived1 extends Base {}
class Derived2 extends Base {}

let x: Base = ...
let y: Derived1 | Derived2 = ...

x = y // OK, both Derived1 and Derived2 are subtypes of Base
y = x // Compile-time error

let x: Base | string = ...
let y: Derived1 | string ...
x = y // OK, Derived1 is subtype of Base
y = x // Compile-time error
```

2. Type `T` is a subtype of union type `U` (U_1 | ... | U_n) if for some `i` `T` is a subtype of U_i.

```
let u: number | string = 1 // OK
u = "aa" // OK
u = 1.0  // OK, 1.0 is of type 'number' (double)
u = 1    // Compile-time error, type 'int' is not a subtype of 'number'
u = true // Compile-time error
```

Note

If union type normalization produces a single type, then this type is used instead of the initial set of union types. This concept is represented in the example below:

```
let u: "abc" | "cde" | string // type of 'u' is string
```

Note

A case of two union types is clarified as follows: union type `U2` ($U2_1$ `|` `...` `|` $U2_n$) is a subtype of union type `U1` ($U1_1$ `|` `...` `|` $U1_m$) if Step 1 applies first, followed by Step 2 applied to every type of `U2`.

```
class T1 {}
class T2 {}
class T3 extends T1 {}  // T3 <: T1
class T4 extends T2 {}  // T4 <: T2
class T5 {}

type U1 = T1 | T2
type U2 = T3 | T4 | T5

function foo (u1: U1, u2: U2) {
    u1 = u2
    // step 1. U2 to be a subtype of U1 iff T3 <: U1 and T4 <: U1 and T5 <: U1
    // step 2.
    //         T3 to be a subtype of T1 or T2 - T1 true
    //         T4 to be a subtype of T1 or T2 - T2 true
    //         T5 to be a subtype of T1 or T2 - false for both
    // Compile-time error as a result
}
```

15.2.6 Subtyping for Function Types

Function type `F` with parameters FP_1 `,` `...` `,` FP_m, rest parameter `FPrest` (if present) and return type `FR` is a *subtype* of function type `S` with parameters SP_1 `,` `...` `,` SP_n, rest parameter `SPrest` (if present) and return type `SR` if **all** of the following conditions are met:

- Number of optional and required parameters of `F` (`m`) is equal or less than number of optional and required parameters of `S` (`n`).
- Number of required parameters of `F` is equal or less than number of required paramers of `S`.

- If the rest parameter `FPrest` is present then `SPrest` is present.
- For each `i <= m`, type of SP_i is a subtype of type of FP_i.
- If the rest parameter `FPrest` is present:
  - For each `i > m`, parameter type `SP` must be a subtype of the element type of type of `FPrest`.
  - Type of `SPrest` should be a subtype of type of `FPrest`.
- The resultant type `FR` is a subtype of `SR`.

```
class Base {}
class Derived extends Base {}

function check(
   bb: (p: Base) => Base,
   bd: (p: Base) => Derived,
   db: (p: Derived) => Base,
   dd: (p: Derived) => Derived
) {
   bb = bd
   /* OK: identical parameter types, and covariant return type */
   bb = dd
   /* compile-time error, parameter types are not contravariant */
   db = bd
   /* OK: contravariant parameter types, and covariant  return type */

   let f: (p: Base, n: number) => Base = bb
   /* OK: subtype has less parameters */

   let g: () => Base = bb
   /* compile-time error, less parameters than expected */
}

let foo: (x?: number, y?: string) => void = (): void => {} // OK: ``m <= n``
foo = (p?: number): void => {}                             // OK:  ``m <= n``
foo = (p1?: number, p2?: string): void => {}               // OK: Identical types
foo = (p: number): void => {}
      // Compile-time error, 1st parameter in type is optional but mandatory in lambda
foo = (p1: number, p2?: string): void => {}
      // Compile-time error,  1st parameter in type is optional but mandatory in lambda
```

15.2.7 Subtyping for Fixed-Size Array Types

If a fixed-size array contains elements of reference types (see *Reference Types*), then the subtyping of elements is based on the subtyping of element types. It is formally described as follows:

`FixedSize<B> <: FixedSize<A>` if `B <: A`.

The situation is represented in the following example:

```
let x: FixedArray<string> = ["aa", "bb", "cc"]
let y: FixedArray<Object> = x // OK, as string <: Object
x = y // Compile-time error
```

The subtyping allows array assignments that can cause an `ArrayStoreError` at runtime if a value of a type that is not a subtype of the element type of one array is put into that array by using a subtype of the element type of another array. Type safety is ensured by runtime checks performed by the runtime system as represented in the example below:

```
class C {}
class D extends C {}

function foo (ca: FixedArray<C>) {
  ca[0] = new C() // ArrayStoreError if ca refers to FixedArray<D>
}

let da: FixedArray<D> = [new D()]

foo(da) // leads to runtime error in 'foo'
```

15.2.8 Subtyping for Difference Types

Difference type `A - B` is a subtype of `T` if `A` is a subtype of `T`.

Type `T` is a subtype of the difference type `A - B` if `T` is a subtype of `A`, and no value belongs both to `T` and `B` (i.e., `T & B = never`).

15.3 Type Identity

Identity relation between two types means that the types are indistinguishable. Identity relation is symmetric and transitive. Identity relation for types `A` and `B` is defined as follows:

- Array types `A = T1[]` and `B = Array<T2>` are identical if `T1` and `T2` are identical.
- Tuple types $A = [T_1, T_2, \ldots, T_n]$ and $B = [U_1, U_2, \ldots, U_m]$ are identical on condition that:
  - n is equal to m, i.e., the types have the same number of elements;
  - Every T_i is identical to U_i for any i in $1 \ldots n$.
- Union types $A = T_1 \mid T_2 \mid \ldots \mid T_n$ and $B = U_1 \mid U_2 \mid \ldots \mid U_m$ are identical on condition that:
  - n is equal to m, i.e., the types have the same number of elements;
  - U_i in `U` undergoes a permutation after which every T_i is identical to U_i for any i in $1 \ldots n$.
- Types `A` and `B` are identical if `A` is a subtype of `B` (`A<:B`), and `B` is at the same time a subtype of `A` (`A:>B`).

Note

Type Alias Declaration creates no new type but only a new name for the existing type. An alias is indistinguishable from its base type.

Note

If a generic class or an interface has a type parameter T while its method has its own type parameter T, then the two types are different and unrelated.

```
class A<T> {
   data: T
   constructor (p: T) { this.data = p } // OK, as here 'T' is a class type parameter
   method <T>(p: T) {
       this.data = p // Compile-time error as 'T' of the class is different from 'T' of␣
↪the method
   }
}
```

15.4 Assignability

Type T_1 is assignable to type T_2 if:

- T_1 is a subtype of T_2 (see *Subtyping*); or
- *Implicit conversion* (see *Implicit Conversions*) is present that allows converting a value of type T_1 to type T_2.

Assignability relationship is asymmetric, i.e., that T_1 is assignable to T_2 does not imply that T_2 is assignable to type T_1.

15.5 Invariance, Covariance and Contravariance

Variance is how subtyping between types relates to subtyping between derived types, including generic types (See *Generics*), member signatures of generic types (type of parameters, return type), and overriding entities (See *Override-Compatible Signatures*). Variance can be of three kinds:

- Covariance,
- Contravariance, and
- Invariance.

Covariance means it is possible to use a type which is more specific than originally specified.

Contravariance means it is possible to use a type which is more general than originally specified.

Invariance means it is only possible to use the original type, i.e., there is no subtyping for derived types.

Valid and invalid usages of variance are represented in the examples below. If class `Base` is defined as follows:

```
class Base {
   method_one(p: Base): Base {}
   method_two(p: Derived): Base {}
   method_three(p: Derived): Derived {}
}
```

—then the code below is valid:

```
class Derived extends Base {
   // invariance: parameter type and return type are unchanged
   override method_one(p: Base): Base {}

   // covariance for the return type: Derived is a subtype of Base
   override method_two(p: Derived): Derived {}

   // contravariance for parameter types: Base is a supertype for Derived
   override method_three(p: Base): Derived {}
}
```

On the contrary, the following code causes compile-time errors:

```
class Derived extends Base {

   // covariance for parameter types is prohibited
   override method_one(p: Derived): Base {}

   // contravariance for the return type is prohibited
   override method_three(p: Derived): Base {}
}
```

15.6 Compatibility of Call Arguments

The following semantic checks must be performed to arguments from the left to the right when checking the validity of any function, method, constructor, or lambda call:

Step 1: All arguments in the form of a spread expression (see *Spread Expression*) that spreads an array literal (see *Array Literal*) are to be linearized recursively to ensure that no spread expression of that form remains at the call site.

Step 2: The following checks are performed on all arguments from left to right, starting from `arg_pos` = 1 and `par_pos` = 1:

if parameter at position `par_pos` is of non-rest form then

if $T_{\text{arg_pos}}$ <: $T_{\text{par_pos}}$ then increment `arg_pos` and `par_pos` else a compile-time error occurs, exit Step 2

else // parameter is of rest form (see *Rest Parameter*)

if parameter is of rest_array_form then

if argument is a spread expression of an array of some type *U* and 'U' <: $T_{\text{rest_array_type}}$ then increment `arg_pos` else if $T_{\text{arg_pos}}$ <: $T_{\text{rest_array_type}}$ then increment `arg_pos` else increment `par_pos`

else // parameter is of rest_tuple_form

for *rest_tuple_pos* in 1 .. rest_tuple_types.count do

if $T_{\text{arg_pos}}$ <: $T_{\text{rest_tuple_pos}}$ then increment `arg_pos` and *rest_tuple_pos* else if rest_tuple_pos < rest_tuple_types.count then increment `rest_tuple_pos` else a compile-time error occurs, exit Step 2

end increment `par_pos`

end

end

The checks are represented in the examples below:

```
 function call (...p: Object[]) {}
 call (...[1, "str", true], ...[ 123])  // Initial call form
 call (1, "str", true, 123)             // To be unfolded into the form with
                                        // no spread expressions

let arr: number[] = [1, 2, 3]
tricky_call (...arr)  // Type 'number' is assignable to 'Object', and a new
                      // array of type 'Object[]' is created from elements
                      // of the array of numbers

function tricky_call (...p: Object[]) {
   p[0] = true
   console.log ("Initial array: ", arr)
   console.log ("Parameter array: ", p)
}

 function foo1 (p: Object) {}
 foo1 (1)  // Type of '1' must be assignable to 'Object'
           // p becomes 1

 function foo2 (...p: Object[]) {}
 foo2 (1, "111")  // Types of '1' and "111" must be assignable to 'Object'
           // p becomes array [1, "111"]

 function foo31 (...p: (number|string)[]) {}
 foo31 (...[1, "111"])  // Type of array literal [1, "111"] must be assignable to␣
↪(number|string)[]
           // p becomes array [1, "111"]

 function foo32 (...p: [number, string]) {}
 foo32 (...[1, "111"])  // Types of '1' and "111" must be assignable to 'number' and 'string
↪' accordingly
           // p becomes tuple [1, "111"]

 function foo4 (...p: number[]) {}
 foo4 (1, ...[2, 3])  //
           // p becomes array [1, 2, 3]
```

15.7 Type Inference

ArkTS supports strong typing but allows not to burden a programmer with the task of specifying type annotations everywhere. A smart compiler can infer types of some entities and expressions from the surrounding context. This technique called *type inference* allows keeping type safety and program code readability, doing less typing, and focusing on business logic. Type inference is applied by the compiler in the following contexts:

- *Type Inference for Constant Expressions*;
- Variable and constant declarations (see *Type Inference from Initializer*);
- Implicit generic instantiations (see *Implicit Generic Instantiations*);
- Function, method or lambda return type (see *Return Type Inference*);
- Lambda expression parameter type (see *Lambda Signature*);
- Array literal type inference (see *Array Literal Type Inference from Context*, and *Array Type Inference from Types of Elements*);
- Object literal type inference (see *Object Literal*).

15.7.1 Type Inference for Constant Expressions

The type of a constant expression (see *Constant Expressions*) is first inferred from the context, if the context allows. The following contexts allow inference:

- *Assignment-like Contexts*;
- *Cast Expression* context;
- *Numeric Operator Contexts*.

If context does not allow to to infer type, the *value default type* is set as follows:

- `int` for a constant expression of a *big integer type* (see *Specifics of Constant Expressions Evaluation*) if its value can be represented by a 32-bit number;
- `long` for a constant expression of a *big integer type* (see *Specifics of Constant Expressions Evaluation*) if its value can be represented by a 64-bit number;
- `double` or `float` for a floating-point constant expression (see *Specifics of Constant Expressions Evaluation*).

Type inference is used only where the *target type* is one of the following:

- Case 1: *Numeric type* (see *Numeric Types*); or
- Case 2: Union type (see *Union Types*) containing at least one *numeric type*.

Case 1: Target type is a numeric type

Where a *target type* is numeric, the type of a constant expression is inferred from the *target type* and the value of the const expression, if one of following conditions is met:

1. *Target type* is equal to or larger than the value default type; or
2. Value is an *integer value* that fitting into the range of the *target type*.

> **Note**
>
> A *floating-point suffix* if present declares a `floating-point literal`, and no type inference occurs.

If none of the conditions above is met, then the *target type* is not a valid type for the constant expression, and a compile-time error occurs.

Type inference for a numeric *target type* is represented in the examples below:

```
let l: long = 1     // OK, target type is larger than literal default type
let b: byte = 127   // OK, integer value fits type 'byte'
let f: float = 123f // OK, target type is the same as literal default type
let g: double = 11  // OK, target type is larger than literal default type

b = 63 + 64           // OK, integer value fits type 'byte'
b = 1 as short        // OK, integer value fits type 'byte'
let s: short = -32767 // OK, integer value fits type 'short'

l = 1.0    // Compile-time error, 'double' cannot be assigned to 'long'
b = 128    // Compile-time error, value is out of range
f = 3.14   // Compile-time error, 'double' cannot be assigned to 'float'
```

Case 2: Target type is a union type containing at least one numeric type

In the case the *target type* is a union type (N_1 | ... | N_k | ... | U_n), where $k \geq 1$ and N_i is a numeric type, then the inferred literal type is determined as follows:

1. If no N_i suits the value, then the default type is used;
2. If only a single N_i suits the value, then N_i is the inferred type;
3. If several N types suit the value, then the following checks are performed:
   - If the value is an integer value, and only one suitable *integer* N_i is present, then this N_i is the inferred type.

Otherwise, a compile-time error occurs due to type inference ambiguity.

Type inference for a union target type is represented in the examples below:

```
let b: byte | Object = 127 // inferred type for 127 is 'byte'
b = 128 /* inferred type for 128 is 'int' (default literal type),
           which is a subtype of Object
        */

let c: byte | string = 127 // inferred type for 127 is 'byte'
c = 128 /* inferred type for 128 is 'int' (default literal type),
           compile-time error as no type in a union is a supertype for
           'int'
        */

let d: int | double = 1.0 // inferred type for 1.0 is 'double'
d = 1 // inferred type for 1 is 'int'

let e: byte | long = 128 // inferred type for 128 is 'long'
e = 127 // Compile-time error, type inference ambiguity for 127

let f: float | double = 3.14 // inferred type is 'double'
```

(continues on next page)

(continued from previous page)

```
f = 2f                          // inferred type is 'float'

let g: float | double = 3.4e39 // inferred type is 'double'
g = 1 // Compile-time error, type inference ambiguity for 1
```

Note

If *target type* is a union type that contains no numeric type, then the default type of the literal is used. The following code is valid as numeric types are subtypes of 'Object':

```
let x: Object | string = 1 // OK, instance of type 'int' is assigned to 'x'

x = 1.0 // OK, instance of type 'double' is assigned to 'x'
```

15.7.2 Return Type Inference

A missing function, method, getter, or lambda return type can be inferred from the function, method, getter, or lambda body. A compile-time error occurs if return type is missing from a native function (see *Native Functions*).

The current version of ArkTS allows inferring return types at least under the following conditions:

- If there is no return statement, or if all return statements have no expressions, then the return type is `void` (see *Type void or undefined*). It effectively implies that a call to a function, method, or lambda returns the value `undefined`.
- If there are *k* return statements (where *k* is 1 or more) with the same type expression *R*, then R is the return type.
- If there are *k* return statements (where *k* is 2 or more) with expressions of types T_1, ..., T_k, then R is the *union type* (see *Union Types*) of these types (T_1 | ... | T_k), and its normalized version (see *Union Types Normalization*) is the return type. If at least one of return statements has no expression, then type `undefined` is added to the return type union.
- If a lambda body contains no return statement but at least one throw statement (see *throw Statements*), then the lambda return type is `never` (see *Type never*).
- If a function, a method, or a lambda is `async` (see *Asynchronous execution*), a return type is inferred by applying the above rules, and the return type T is not `Promise`, then the return type is assumed to be `Promise<T>`.

Return type inference is represented in the example below:

```
// Explicit return type
function foo(): string { return "foo" }

// Implicit return type inferred as string
function goo() { return "goo" }

class Base {}
class Derived1 extends Base {}
class Derived2 extends Base {}

```

(continues on next page)

(continued from previous page)

```
function bar (condition: boolean) {
    if (condition)
        return new Derived1()
    else
        return new Derived2()
}
// Return type of bar will be Derived1|Derived2 union type

function boo (condition: boolean) {
    if (condition) return 1
}
// That is a compile-time error as there is an execution path with no return
```

15.8 Overriding

Method overriding is the language feature closely connected with inheritance. It allows a subclass or a subinterface to offer a specific implementation of a method already defined in its supertype optionally with modified signature.

The actual method to be called is determined at runtime based on object type. Thus, overriding is related to *runtime polymorphism*.

> **Note**
>
> *Overriding* does not apply to constructors.

ArkTS uses the *override-compatibility* rule to check the correctness of overriding. The *overriding* is correct if method signature in a subtype (subclass or subinterface) is *override-compatible* with the method defined in a supertype (see *Override-Compatible Signatures*).

An implementation is forced to *Make a Bridge Method for Overriding Method* in some cases of *method overriding*.

15.8.1 Overriding in Classes

> **Note**
>
> Only accessible (see *Accessible*) methods are subjected to overriding. The same rule applies to accessors in case of overriding.

An overriding method can retain access modifier of a method from a superclass or a superinterface, or change *protected* for *public* (see *Access Modifiers*). Otherwise, a compile-time error occurs.

```
class Base {
   public public_member() {}
   protected protected_member() {}
   private private_member() {}
}
```

(continues on next page)

(continued from previous page)

```

interface Interface {
   public_member()             // All members are public in interfaces
   private private_member() {} // Except private methods with default implementation
}

class Derived extends Base implements Interface {
   public override public_member() {}
      // Public member can be overridden and/or implemented by the public one
   public override protected_member() {}
      // Protected member can be overridden by the protected or public one
   override private_member() {}
      // A compile-time error occurs as private methods of a superclass or
      // a superinterface are not accessible in the derived class, and such
      // a declaration attempt has nothing to override.
   private_member() {}
      // Will be a correct method declaration which is not related to
      // private methods with the same name and signature from a supoer class
      // or superinterfaces
}
```

If an *instance method* is defined or inherited by a subclass with the same name as the *instance method* in a superclass or superinterface, then the following semantic rules are applied:

- If signatures are not *override-compatible* (see *Override-Compatible Signatures*), and if signatures formed by using *effective signature types* of original signatures are *override-compatible* after type erasure, then a compile-time error occurs.
- If signatures are *override-compatible* (see *Override-Compatible Signatures*), then the method of subinterface overrides the method of superinterface *in* the subinterface.
- Otherwise, *Implicit Method Overloading* is used.

Overriding methods from a superclass is represented in the example below:

```
class Base {
   method_1() {}
   method_2(p: number) {}
}
class Derived extends Base {
   override method_1() {} // overriding
   override method_2(p: string) {} // Compile-time error, not override-compatible
}
```

Overriding a method from a superinterface by a method from a superclass is represented in the example below:

```
interface I {
   m(): void
}

class Base {
   m() { }
}
class Derived extends Base implements I {
```

(continues on next page)

(continued from previous page)

```
    // method 'm' inherited from 'Base' overrides 'm' defined in 'I'
}
```

A single method in a subclass can override several methods of a superclass:

```
class B {}
class C {
    foo(a: A) {}
    foo(b: B) {}
}
class D extends C {
    foo(o: Object) { console.log("foo in D")}
}

let c: C = new D()
c.foo(new A()) // output: foo in D
c.foo(new B()) // output: foo in D
```

If more than one method of the subclass overrides the same method of the superclass a compile-time error occurs:

```
 class I{
    foo (p: C) {}
 }
 class C extends I { // More than one method of class C overrides the same method
    foo (p: C) {}
    foo (p: I) {}
 }
```

15.8.2 Overriding in Interfaces

If a method is defined in a subinterface with the same name as an accessible method in the superinterface, then the following semantic rules apply:

- If signatures are not *override-compatible* (see *Override-Compatible Signatures*), and if signatures formed by using *effective signature types* of original signatures are *override-compatible* after type erasure, then a compile-time error occurs.
- If signatures are *override-compatible* (see *Override-Compatible Signatures*), then the method of subinterface overrides the method of superinterface *in* the subinterface.
- Otherwise, *Implicit Method Overloading* is used.

```
interface Base {
   m(p: string): void
   m(p: number): void
}
interface Derived extends Base {
   m(p: object): void  // m overrides both Base.m(string) and Base.m(number)
}
```

> **Note**
>
> Several methods of superinterface can be overridden by a single method in a subinterface.

```
interface Base {
   method_1()
   method_2(p: number)
   method_3(): string
   method_4(a: Array<string>)
   private foo() {} // private method with implementation body
}
interface Derived extends Base {
   method_1()           // overriding
   method_2(p: string)  // overloading
   method_3(): number // Compile-time error, bad overriding, return type mismatch
   method_4(a: Array<number>)  // Compile-time error, original signatures are
                               // not override-compatible, but effective
                               // signatures after type erasure are compatible.
   foo(p: number): void // it is just a new method declaration
                        // Base.foo() is not accessible here at all
}
```

If more than one method of the subinterface overrides the same method of the superinterface a compile-time error occurs.

```
  interface I{
     foo (p: C): void
  }
  interface C extends I { // More than one method of interface C overrides the same␣
↪method
     foo (p: C): void
     foo (p: I): void
  }
```

15.8.3 Override-Compatible Signatures

Let's consider classes `Derived` and `Base` for which:

- `Derived` is a subclass of `Base`
- both classes declare method `foo`
- in `Base`, `foo()` has a signature S_1 `<`V_1 `,` ... V_k`>` `(`U_1 `,` ..., U_n`)` `:`U_{n+1},
- in `Derived`, `foo()` has a signature S_2 `<`W_1 `,` ... W_j`>` `(`T_1 `,` ..., T_m`)` `:`T_{m+1}

as in the example below:

```
class Base {
   foo <V1, ... Vk> (p1: U1, ... pn: Un): Un+1
}
```

(continues on next page)

(continued from previous page)

```
class Derived extends Base {
   override foo <W1, ... Wj> (p1: T1, ... pm: Tm): Tm+1
}
```

The signature S_2 is override-compatible with S_1 only if **all** of the following conditions are met:

1. Number of parameters of both methods is the same, i.e., `n = m`.
2. Each parameter type T_i is a supertype of U_i for `i` in `1..n` (contravariance).
3. If return type T_{m+1} is `this`, then U_{n+1} is `this`, or any of superinterfaces or superclass of the current type.
4. If return type T_{m+1} is not `this`, then it must be a subtype of U_{n+1} (covariance).
5. Number of type parameters of either method is the same, i.e., `k = j`.
6. Constraints of $W_1, \ldots W_j$ are to be contravariant (see *Invariance, Covariance and Contravariance*) to the appropriate constraints of $V_1, \ldots V_k$.

The following rule applies to generics:

- Derived class must have type parameter constraints to be subtypes (see *Subtyping*) of respective type parameter constraints in the base type;
- Otherwise, a compile-time error occurs.

```
class Base {}
class Derived extends Base {}
class A1 <CovariantTypeParameter extends Base> {}
class B1 <CovariantTypeParameter extends Derived> extends A1<CovariantTypeParameter> {}
    // OK, derived class may have type compatible constraint of type parameters

class A2 <ContravariantTypeParameter extends Derived> {}
class B2 <ContravariantTypeParameter extends Base> extends A2<ContravariantTypeParameter>
↪ {}
    // Compile-time error, derived class cannot have non-compatible constraints of type␣
↪parameters
```

The semantics is represented in the examples below:

1. **Class/Interface Types**

```
interface Base {
    param(p: Derived): void
    ret(): Base
}

interface Derived extends Base {
    param(p: Base): void    // Contravariant parameter
    ret(): Derived          // Covariant return type
}
```

2. **Function Types**

```
interface Base {
    param(p: (q: Base)=>Derived): void
    ret(): (q: Derived)=> Base
}
```

(continues on next page)

(continued from previous page)

```
5
6  interface Derived extends Base {
7      param(p: (q: Derived)=>Base): void  // Covariant parameter type, contravariant␣
   ↪return type
8      ret(): (q: Base)=> Derived          // Contravariant parameter type, covariant␣
   ↪return type
9  }
```

3. **Union Types**

```
1  interface BaseSuperType {}
2  interface Base extends BaseSuperType {
3     // Overriding for parameters
4     param<T extends Derived, U extends Base>(p: T | U): void
5
6     // Overriding for return type
7     ret<T extends Derived, U extends Base>(): T | U
8  }
9
10 interface Derived extends Base {
11    // Overriding kinds for parameters, Derived <: Base
12    param<T extends Base, U extends Object>(
13       p: Base | BaseSuperType // contravariant parameter type:  Derived | Base <: Base␣
   ↪| BaseSuperType
14    ): void
15    // Overriding kinds for return type
16    ret<T extends Base, U extends BaseSuperType>(): T | U
17 }
```

4. **Type Parameter Constraint**

```
1  interface Base {
2      param<T extends Derived>(p: T): void
3      ret<T extends Derived>(): T
4  }
5
6  interface Derived extends Base {
7      param<T extends Base>(p: T): void       // Contravariance for constraints of type␣
   ↪parameters
8      ret<T extends Base>(): T                // Contravariance for constraints of the␣
   ↪return type
9  }
```

Override compatibility with `Object` is represented in the example below:

```
1  enum E  { One, Two }
2  interface Base {
3     kinds_of_parameters<T extends Derived, U extends Base>(
4        // It represents all possible kinds of parameter type
5        p01: Derived,
6        p02: (q: Base)=>Derived,
7        p03: number,
8        p04: T | U,
```

(continues on next page)

(continued from previous page)

```
        p05: E,
        p06: Base[],
        p07: [Base, Base]
    ): void
    kinds_of_return_type(): Object // It can be overridden by all subtypes of Object
}
interface Derived extends Base {
    kinds_of_parameters( // Object is a supertype for all class types
        p1: Object,
        p2: Object,
        p3: Object,
        p4: Object,
        p5: Object,
        p6: Object,
        p7: Object
    ): void
}

interface Derived1 extends Base {
    kinds_of_return_type(): Base // Valid overriding
}
interface Derived2 extends Base {
    kinds_of_return_type(): (q: Derived)=> Base // Valid overriding
}
interface Derived3 extends Base {
    kinds_of_return_type(): number // Valid overriding
}
interface Derived4 extends Base {
    kinds_of_return_type(): number | string // Valid overriding
}
interface Derived5 extends Base {
    kinds_of_return_type(): E // Valid overriding
}
interface Derived6 extends Base {
    kinds_of_return_type(): Base[] // Valid overriding
}
interface Derived7 extends Base {
    kinds_of_return_type(): [Base, Base] // Valid overriding
}
```

15.9 Overloading

Overloading is the language feature that allows using the same name to call several functions, methods, or constructors with different signatures.

The actual function, method, or constructor to be called is determined at compile time. Thus, *overloading* is compile-time *polymorphism by name*.

ArkTS supports the following two *overloading* mechanisms:

- *Implicit overloading*, where a set of overloaded entities for functions and methods is determined by their names, or includes all constructors; and
- *Explicit overloading* (see *Explicit Overload Declarations*) that allows a developer to specify a set of overloaded entities explicitly.

In either case, an ordered set of overloaded entities is built at compile time, and used by *Overload Resolution* to select one entity to call from within the set. *Overload resolution* uses the first-match algorithm to streamline the resolution process, i.e., only the first appropriate entity is called, while all other entities are not considered.

If signatures of *implicitly overloaded entities* are *overload-equivalent*, then a compile-time error occurs (see *Overload-Equivalent Signatures*). *Explicitly overloaded entities* are never checked for overload equivalence. *Explicitly overloaded entities* with separate names never cause a compile-time error.

Overloading for the module scope name *main* is prohibited, and causes a compile-time error if attempted:

```
function main() {}
function main(p: string[]) {}
// Such declarations lead to a compile-time error
```

15.9.1 Implicit Function Overloading

Two or more functions declared with the same name in the same declaration scope are *implicitly overloaded*.

Note

Same-name functions declared in different scopes cannot be *implicitly overloaded* (see *Declaration Distinguishable by Signatures*). A compile-time error occurs if the names of an imported function and of a function declared in the current module are the same.

When calling an overloaded function (see *Function Call Expression*), *Overload Resolution* is used to determine exactly which function must be called.

```
function foo(p: number) {} // #1
function foo(p: string) {} // #2

foo(5)   // function marked #1 is called
foo("5") // function marked #2 is called
```

If signatures of two implicitly overloaded non-generic functions are overload-equivalent (see *Overload-Equivalent Signatures*), then a compile-time error occurs. However, an implicit overload with instantiation of a generic and overload-equivalent non-generic function causes no compile-time error, and the textually first function is used:

```
function goo(x: int): void {}
function goo(x: int): void {}  // Compile-time error, overload-equivalent
                               // non-generic functions

function foo<T>(x: T) {}
function foo(x: number) {}

foo(1)   // OK, instance of foo<T>() with T=number called
```

(continues on next page)

(continued from previous page)

```

function bar(x: number) {}
function bar<T>(x: T) {}

bar(1)   // OK, plain bar() called
```

15.9.2 Implicit Method Overloading

Two or more methods within a class or an interface are implicitly overloaded if:

- All have the same name;
- All are either `static` or non-`static`.

A compile-time error occurs if signatures of two implicitly overloaded methods are *overload-equivalent* (see *Overload-Equivalent Signatures*).

Implicit overload can be caused by method declaration or inheritance:

```
class Base{
    foo(p: number) {} // #1
}

class Derived extends Base {
    foo(p: string) {} // #2
}

let d = new Derived()

d.foo(5)   // method marked #1 is called
d.foo("5") // method marked #2 is called
```

When calling an overloaded method (see *Method Call Expression*), *Overload Resolution* is used to determine exactly which method is to be called.

```
class C{
    foo(p: number) {} // #1
    foo(p: string) {} // #2
}

let c = new C()

c.foo(5)   // method marked #1 is called
c.foo("5") // method marked #2 is called
```

If an instantiation of a generic class or a generic interface leads to an *overload-equivalent* method, then the textually first method is called:

```
 class Template<T> {
    foo (p: T) { console.log("generic foo") }
```

(continues on next page)

(continued from previous page)

```
    foo (p: number) { console.log("plain foo") }
    bar (p: number) { console.log("plain bar") }
    bar (p: T) { console.log("generic bar") }
}

// This instantiation leads to two *overload-equivalent* methods
let instantiation: Template<number> = new Template<number>

instantiation.foo(1)  // prints 'generic foo'
instantiation.bar(1)  // prints 'plain bar'
```

15.9.3 Implicit Constructor Overloading

Two or more constructors within a class are *implicitly overloaded*. If signatures of two overloaded constructors are *overload-equivalent* (see *Overload-Equivalent Signatures*), then a compile-time error occurs.

Overload Resolution is used to determine exactly which one constructor is to be called in a class instance creation expression (see *New Expressions*).

```
class BigFloat {
    /*other code*/
    constructor (n: number) {/*body1*/} // #1
    constructor (s: string) {/*body2*/} // #2
}

new BigFloat(1)      // constructor, marked #1,  is used
new BigFloat("3.14") // constructor, marked #2,  is used
```

15.9.4 Overload-Equivalent Signatures

Signatures *S1* and *S2* are *overload-equivalent* if:

- Both have the same number of parameters;
- *Effective signature types* (see *Type Erasure*) of each parameter type in *S1* and *S2* are equal.

Return types of *S1* and *S2* do not affect *overload-equivalence*.

The following code causes a compile-time error as function signatures are *overload-equivalent*:

```
function foo(x: Array<string>): string {}
function foo(x: Array<number>): number {} // Compile-time error
```

15.10 Overload Resolution

Overload resolution is used to select one entity to call from an *ordered set of overloaded entities* (called in short *overload set*) in a function call or a method call or a constructor call, where the constructor call is a part of a new expression (see *New Expressions*).

An *overload set* for each overloaded name is formed while processing declarations. The process (see *Forming an Overload Set*) takes into account the following:

- Textual order of implicitly overloaded entities;
- Listing order of explicit overload declarations; and
- Inheritance.

The *overload resolution* process is rather straightforward. The process takes overloaded entities one after another from an *overload set*, and checks the call of each entity to be valid for the arguments given. The first entity for which the call is valid is the selected entity. Other entities are not checked after the first valid entity is found.

A compile-time occurs if a call is invalid for any overload entities.

A call of an entity is valid if:

- Each required parameter has an argument;
- There is no excess argument that fails to correspond to a parameter, including to an optional parameter or to a rest parameter;
- Each argument is assignable to a corresponding parameter type, or to a corresponding element type of a rest parameter.

Return types of overload entities are not considered by *overload resolution*, it means that the selected entity can lead to a compile-time error, like in the following example:

```
function foo(n: number): number {}
function foo(s: string): string {}

let x: number = foo("1") // Compile-time error
```

15.10.1 Forming an Overload Set

Only a single *overload set* is created for each overloaded name in a scope. An overload set is formed in exactly one of the following ways:

1. If a call uses a name that denotes an *explicit overload declaration*, then the overload set consists of the entities listed in that declaration, in the listed order.
2. Otherwise, if a call uses a name that denotes *implicit overloading*, then the overload set consists of the implicitly overloaded entities in textual declaration order.

A compile-time error occurs if an identifier in an explicit overload declaration denotes an implicitly overloaded name rather than one accessible entity.

The textual position of an *explicit overload* does not influence the order in the *overload set*. An *explicit overload* is effectively processed at the end of the scope.

An overload set for interface and instance class methods can contain methods from superinterfaces and superclasses. Methods defined in an interface or in a class are added to the *overload set* before any inherited method, i.e., more specific

entities have a higher priority. Further details are discussed in *Overload Set for Interface Methods* and *Overload Set for Class Instance Methods*.

15.10.2 Overload Set for Functions

For a given function name, the overload set is formed as follows:

- If the name denotes an *explicit function overload*, then the overload set is the ordered list declared by that overload.
- Otherwise, if the name denotes *implicit function overloading*, then the overload set consists of the same-name functions in textual declaration order.

No overload set for functions combines an explicit overload declaration with an implicit overload set. The examples below illustrate both forms:

```
function foo() {}           // implicitly overloaded foo#1
function foo(b: boolean) {} // implicitly overloaded foo#2
// The overload set for 'foo' is {foo#1, foo#2}

function foo0() {}
function foo1(b: boolean) {}
overload bar {foo0, foo1}
// The overload set for 'bar' is {foo0, foo1}

foo()     // foo#1 is called
foo(true) // foo#2 is called
bar()     // foo0 is called
bar(true) // foo1 is called
```

15.10.3 Overload Set for Interface Methods

An overload set for a given interface is formed from the following:

- Either implicitly overloaded methods (see *Implicit Method Overloading*) declared in the interface, or explicitly overloaded methods listed in *Explicit Interface Method Overload*;
- Overload sets from superinterfaces, if any.

The following steps are taken to form an overload set:

1. The local overload set for a given name is formed in the interface by using either implicit overloading or an explicit overload declaration, in the order described in *Forming an Overload Set*.
2. Overload sets from each direct superinterface are added at the end of an overload set in the order of the `implements` clauses. A method that is already added to the overload set is not added again.

Note

Overload sets from non-direct superinterfaces are not considered as they are already accounted for in direct superinterfaces.

Forming an *overload set* for an interface that has no superinterface is represented in the example below:

```
interface I {
    foo(x: number) // #1
    foo(s: string) // #2
    // The overload set for 'foo' is {foo#1, foo#2}
}

interface J {
    fooNum(x: number)
    fooStr(s: string)
    overload foo { fooNum, fooStr }
    // The overload set for 'foo' is {fooNum, fooStr}
}
```

Overload sets for an interface with superinterfaces is represented in the example below:

```
interface I1 {
    foo(x: number)  // #1
}
interface I2 {
    foo(s: string)  // #2
    foo(b: boolean) // #3
}
interface J extends I1, I2 { // the order in extends list is used
    foo() // #4
    /* The overload set is {foo#4, foo#1, foo#2, foo#3}
       Formed as: set(J)={foo#4} append set(I1)={foo#1} append set(I2)={foo#2, foo#3}
    */
}
```

That overridden methods occur only once in a list (`I` and `I2` as defined above) is represented in the example below:

```
interface K extends I1, I2 {
    foo(s: string) // #2 as it overrides I2.foo
    /* The overload set is {foo#2, foo#1, foo#3}
       Formed as: set(K)={foo#2} append set(I1)={foo#1} append set(I2)={foo#2, foo#3}
       Second occurrence of foo#2 is skipped.
    */
}
```

15.10.4 Overload Set for Class Static Methods

An overload set of static methods for a given class is formed as follows:

- If the name denotes an *explicit class method overload*, then the overload set is the ordered list declared by that overload.

- Otherwise, if the name denotes *implicit method overloading*, then the overload set consists of the same-name static methods in textual declaration order.

The algorithm that defines the order of an *overload set* considers only the static methods defined directly in a class scope (see *Forming an Overload Set*) because static methods are not inherited.

Static explicit and static implicit overloading do not combine for one name. The formation and the use of an *overload set* by the *overload resolution* is represented in the example below:

```
class C {

    static foo0() {}
    static foo1(b: boolean) {}
    static overload foo {foo0, foo1}
    // The overload set for 'foo' is {foo0, foo1}
}

C.foo()     // foo0 is called
C.foo(true) // foo1 is called
```

15.10.5 Overload Set for Class Instance Methods

An overload set for class instance methods of a given class is formed from the following:

- Either implicitly overloaded methods (see *Implicit Method Overloading*) declared in the class, or explicitly overloaded methods listed in *Explicit Class Method Overload*;
- Methods from a direct superclass, if any.

The following steps are taken to form an overload set:

1. The local overload set for a given name is formed in the class by using either implicit overloading or an explicit overload declaration, in the order described in *Forming an Overload Set*, including the methods that override or implement methods from supertypes.
2. Overload set from a direct superclass (if any) is added at the end of an overload set. A method that is already added to the overload set is not added again.
3. Overload sets from each superinterface are added at the end of an overload set in the order of the `implements` clauses. A method that is already added to the overload set is not added again.

> **Note**
>
> Overload sets from non-direct supertypes are not considered.

Forming an *overload set* for a class that has neither a superclass nor a superinterface is represented in the example below:

```
class C {
    foo(x: number) {} // #1
    foo(s: string) {} // #2
    // The overload set for 'foo' is {foo#1, foo#2}
```

(continues on next page)

(continued from previous page)

```
}

class D {
    fooNum(x: number) {}
    fooStr(s: string) {}
    overload foo { fooNum, fooStr }
    // The overload set for 'foo' is {fooNum, fooStr}
}
```

An overload set for a class with a superclass and a superinterface is represented in the example below:

```
interface I {
    foo(x: number)  // #1
}

class C {
    foo(s: string) {} // #2
}

class D extends C implements I{
    foo(x: number) {}  // overrides #1
    foo(x: boolean) {} // #3
    /* The overload set is {foo#1, foo#3, foo#2}
       Formed as: set(D)={foo#1, foo#3} append set(C)={foo#2} append set(I)={foo#1}
       Second occurrence of foo#1 is skipped.
    */
}
```

Only direct supertypes are considered for overload sets. It is represented in the example below:

```
interface I{
    foo(x: number) {} // #1
}
class C implement I{
    foo(x: A)      // #2
    // Note: foo in I has default body, no need to implement it in C
    // The overload set is {foo#2, foo#1}
}
class D extends C {
    foo(s: string) {}   // #3
    foo(x: A | undefined) {} // overrides #2
    /* The overload set is {foo#3, foo#2, foo#1}
       Formed as: set(D)={foo#3, foo#2} append set(C)={foo#2, foo#1}
       Second occurrence of foo#2 from set(C) is skipped.
    */
}
class E extends D {
    foo(x: number) {} // overrides #1
    /* The overload set is {foo#1, foo#3, foo#2}
       Formed as: set(E)={foo#1} append set(D)={foo#3, foo#2, foo#1}
       Second occurrence of foo#1 from set(D) is skipped.
    */
}
```

More details are provided in *Overloading and Overriding*.

15.10.6 Overload Set for Constructors

For a given class, the overload set for constructors is formed from implicitly overloaded constructors (see *Implicit Constructor Overloading*). The order of constructors in the overload set is textual order of constructor declarations.

The example below illustrates how *overload set* is formed by *overload resolution*:

```
class C {
    constructor () {}       // ctor#1
    constructor (s: string) {} // ctor#2
}
/* The overload set of constructors for class 'C' is
   {ctor#1, ctor#2} */

new C()     // ctor#1 is used
new C("aa") // ctor#2 is used
```

15.10.7 Overload Set Warning

A compile-time warning is issued if the order of entities in an overload set implies that some overloaded entity can never be selected for a call:

```
function f1 (p: number) {}
function f2 (p: number|string) {}
function f3 (p: string) {}
overload foo {f1, f2, f3}  // warning: f3 will never be called as foo()

foo (5)                    // f1() is called
foo ("5")                  // f2() is called
```

15.10.8 Overload Set at Method Call

Additional processing of an *overload set* is used at *Method Call Expression* because an identifier at the call site can denote both *instance methods* and *Functions with Receiver*.

If the identifier at the call expression denotes *functions with receiver* only, then *Overload Set for Functions* is used. However, only *functions with receiver* (not ordinary functions) are considered for *overload resolution*:

```
class C {}

function foo(this: C) {}            // #1
```

(continues on next page)

(continued from previous page)

```
function foo(this: C, s: string) {} // #2
function foo(c: C, n: number) {}    // #3

let c = new C()
c.foo() // only function #1, #2, but not #3 are considered here

foo(c) // all three functions are considered here
```

If the identifier denotes both instance methods and *functions with receiver*, then the overload set for methods is used for *overload resolution*, i.e., no function with receiver is called. Otherwise, a compile-time warning is issued as represented in the example below:

```
// File1
export class C {
    bar() {}
}
export function foo(this: C) {}

// File2
import {C, foo as bar} from "File1"

new C().bar() // C.bar is called, warning is issued
```

Note

If a function with receiver is declared, and the name of the function is the same as the name of an accessible instance method or field of the receiver type, then a compile-time error occurs in most cases (see the example in *Functions with Receiver*). A warning is issued where no such error is reported.

15.10.9 Overloading and Overriding

When calling an overloaded method (class instance method or interface method), both *Overloading* and *Overriding* influence the actual method to call. As *compile-time* polymorphism, *overload resolution* selects the method to call from a class type or an interface type known at compile time. However, the method can be overridden in subtypes, and the actual method is called in accordance with the *runtime type* of the object from which the method is called.

Note

Overriding does not influence forming of overload sets by itself despite any method declared in a class—both new or overridden—is placed in the overload set before any inherited method.

The manner *overloading* and *overriding* influence the method to call is represented in the example below:

```
class C1 {}
class C2 extends C1 {}

```

(continues on next page)

(continued from previous page)

```
class A {
    foo(c: C2) { console.log("A.C2") }
    foo(c: C1) { console.log("A.C1") }
}

class B extends A {
    override foo(c: C2) { console.log("B.C2") }
}

let a: A = new B()
a.foo(new C2()) // 1st call output: B.C2
a.foo(new C1()) // 2nd call output: A.C1
```

The details of the first call are as follows:

- Static type of `a` is `A`, and only this type is considered for overload resolution;
- First overloaded `foo` can be called, and is selected;
- Runtime type of `a` is `B`, `foo(c: C2)` is overridden in `B`, and then the method from `B` is called.

The details of the second call are as follows:

- `foo(c: C1)` is selected to call;
- `foo(c: C1)` is not overridden, and the original method from `A` is called.

The situation where a single method in a subclass overrides several methods of a superclass is represented in the example below:

```
class C1 {}
class C2 extends C1 {}

class A {
    foo(c: C2) { console.log("A.C2") }
    foo(c: C1) { console.log("A.C1") }
}

class D extends A {
    foo(c: C1) { console.log("D.C1") }
}

let a: A = new D()
a.foo(new C2()) // 1st call output: D.C1
a.foo(new C1()) // 2nd call output: D.C1
```

In the example above, `foo` in `D` overrides both `A` methods (see *Override-Compatible Signatures*). As a result, the same method is called despite different methods are selected at compile time for the first and the second calls.

15.10.10 Dynamic resolution of method calls

The actual method to be invoked during the *Method Call Expression* evaluation is determined in the runtime with respect to the method statically resolved during the compile time (see *Overload Set at Method Call*) and the actual *type* of the `objectReference`.

The resolution depends on the form of the call expression:

- For *static method calls*, overriding is not used and the statically determined method is the one to be invoked
- For *super calls*, overriding is not used and the statically resolved method of superclass is the one to be invoked
- For *instance method calls*, the actual type of the `objectReference` referred to as *T* is used to determine the method to be invoked.

For the statically resolved method *M* defined in the type *D*, let the type *C* be

- *D* if the method *M* is found in the type *D* during the execution.
- The *closest* superclass of *C* that defines the method of the signature of *M*.
- The *closest* superinterface of *C* that defines the method of the signature of *M*.

Note: For the set of programs compiled without source code updates *C* is always *D*

Type *T* (which is always a class) and the statically determined method *M* defined in the type *C* (where *T* is necessarily a subtype of *C*) are used to perform the resolution, which is defined as follows:

- If *T* is *C*, the result of the resolution is *M*.
- If *T* has a superclass and the resolution of the method *M* for the superclass of *T* succeeds and the resolved method is defined in class, let *Ms* be the result of the resolution:
  - If the type *T* defines several method declarations that override the method *Ms* in *T*:
    - If the set of such methods contains exactly one method, this method is the result of the resolution.
    - Otherwise, the method resolution fails for type *T*.
  - Otherwise, *Ms* is the result of the resolution.
- Otherwise, the set of the *superinterfaces* of *T* is searched for a matching method:
  - Each considered method should be declared in the superinterface of *T* and should override the method *M* in *C*.
  - For each considered method *Mi*, there should be no other method *Mio* satisfying the previous criterion that overrides *Mi* in the interface that defines *Mio*.

  Note: That means, all considered method belong to subinterfaces of the declaring interface of *M*.

  If the set contains exactly one default method, this method is the result of the resolution. Otherwise, the set either
  - has multiple default methods
  - has no default methods

  In these cases, the resolution fails for type *T*.

If the method resolution fails, then a runtime error occurs.

Note: For the set of programs compiled without source code updates the resolution always results in method resolved and does never throw.

15.11 Type Erasure

Type erasure is the concept that refers to a special handling of some language *types*, primarily *Generics*, as members of a smaller subset of the language *type system* when considering certain parts of the language semantics. The subset defined via type mapping is referred to as *effective type*. *Effective type* mapping satisfies the following properties:

- For arbitrary types `A` and `B`, if `A` is a subtype of `B`, then an `EffectiveType(A)` is a subtype of `EffectiveType(B)`
- For arbitrary types `A` and `B`, `EffectiveType(A | B)` is identical to `EffectiveType(A) | EffectiveType(B)`
- For an arbitrary type `A`, `A | undefined` is a subtype of `EffectiveType(A | undefined)`
  - In particular, for an arbitrary type `A`, `undefined` is a subtype of `EffectiveType(A)`

An original type and an *effective type* can have relationships of two kinds:

- If *Effective type* of `T` is identical to `T`, then type `T` is *preserved by type erasure*;
- Otherwise, the type is considered as *affected by type erasure*.

If `T | undefined` is *preserved by type erasure*, then type `T` is *preserved up to undefined*.

This property limits the possible applications of type-checking expressions:

- *instanceof Expression*;
- *Cast Expression*.

Type mapping determines *effective types* as follows (the `undefined` union member is excluded in the right-hand-side column for brevity):

Original Type	Effective Type (undefined member excluded)
`Any`	`Any`
`never`	`never`
`undefined`	`undefined`
`null`	`null`
Generic types (see *Generics*)	Instantiation of the same generic type (see *Explicit Generic Instantiations*) with type arguments selected in accordance with *Type Parameter Variance*: • *Covariant* type parameters are instantiated with the constraint type; • *Contravariant* type parameters are instantiated with type `never`;
Type Parameters	*Type Parameter Constraint*
Union Types in the form `T1 | T2 ... Tn`	Union of effective types of each constituent type of `T1 ... Tn`.
Array Types in the form `T[]`	Same as for a generic type `Array<T>`.
Fixed-Size Array Types (`FixedArray<T>`)	Instantiations of `FixedArray<T>` (i.e., the effective type of type argument `T` is preserved).
Function Types in the form `(P1, P2, Pn) => R`	Instantiation of an internal generic *function* type with respect to the number of parameter types *n*. Parameter types `P1, P2 ... Pn` are instantiated with `Any`. Return type `R` is instantiated with type `never`.
Function Types in the form `(P1, P2, Pn, ... PR) => R`	Instantiation of an internal generic *rest-parametrized function* type with respect to the number of parameter types *n*. Internal generic *rest-parametrized function* of *n* parameters is a supertype of the internal generic *function* type of *n* parameters. Parameter types `P1, P2 ... Pn` and rest parameter type `PR` are instantiated with `Any`. Return type `R` is instantiated with type `never`.
Tuple Types in the form `[T1, T2 ..., Tn]`	Instantiation of an internal generic tuple type with respect to the number of element types *n*.
String Literal Types	`string`
Awaited<T>	• If `T` is neither a type parameter nor a subtype of `Promise`, then the Effective type (Awaited<T>) is the Effective type (T); • If `T` is a `Promise<U>`, then the Effective type (Awaited<T>) is the Effective type (U); • If `T` is a type parameter with `in` variance, then the Effective type (Awaited<T>) is `never`; • If `T` is a type parameter with `out` variance or no variance specified, then the Effective type (Awaited<T>) is the Effective type (upper-bound(T)).
NonNullable<T>	Effective type (T) - `null`
Partial<T>	Special runtime partial class
Required<T>	`Effective type (T)`
Readonly<T>	`Effective type (T)`
Record<K, V>	`Map <Effective type (K), Effective type (V)>`
ReturnType<F>	`Effective type (return type of F)`

Additional type mapping defines an *effective signature type*, i.e., an *effective type* of a corresponding type except the

following:

Original Type	Effective signature type
Return type `never`	`never`

Otherwise, the original type is *preserved*.

A program that uses types not *preserved* by erasure, and relies on certain cast expressions (see *Cast Expression*) which narrow values to types not *preserved up to undefined*, can cause `ClassCastError` thrown during the evaluation of *Field Access Expression*, *Method Call Expression*, or *Function Call Expression*. Checks that cause any runtime error mentioned above ensure type safety of program execution:

```
class A<T> {
  field: T

  constructor(value: T) {
    this.field = value
  }
}

function unsafe(p: Object): A<number> {
  return p as A<number>  // OK, but check is performed against type A<>, but not against A<number>
                         // thus it can cause exception later during execution
}

let a: A<number> = unsafe(new A<string>("a")) // A<string> resides in A<number>

let n = a.field // An exception is raised as the expected type is number but the actual type is string
```

15.12 Static Initialization

Static initialization is a routine performed once for each class (see *Classes*), namespace (see *Namespace Declarations*), or module (see *Namespaces and Modules*).

Static initialization presumes executing the following:

- *Initializers* of *variables* or *static fields*;
- *Top-level statements* for modules and namespaces;
- Code inside a *static block* for classes.

Static initialization is performed before one of the following operations is executed for the first time:

- Calling a class static method (see *Method Call Expression*), accessing a class static field (see *Accessing Static Fields*), and creating a new class instance (see *New Expressions*) or a *static initialization* of a direct subclass;
- Calling a function or accessing a variable of a namespace or a module.

> **Note**
>
> None of the operations above invokes a *static initialization* of the same entity recursively if it is not completed.
>
> The code in a static block of a namespace is executed only if namespace members are used in a program (an example is provided in *Namespace Declarations*).

An initialization is not complete if the execution of a *static initialization* is terminated due to an exception thrown. A repeated attempt to execute the *static initialization* can throw an exception again.

If a *static initialization* invokes a concurrent execution (see *Execution model*), then all jobs that try to invoke it are synchronized. The synchronization is to ensure that the initialization is performed only once, and that the operations requiring the *static initialization* to be performed are executed after the initialization completes.

If *static initialization* routines of two concurrently initialized classes are circularly dependent, then a deadlock can occur.

15.12.1 Static Initialization Safety

A compile-time error occurs if a *named reference* refers to a not yet initialized *entity*, including one of the following:

- Variable (see *Variable and Constant Declarations*) of a module or namespace (see *Namespace Declarations*);
- Static field of a class (see *Static and Instance Fields*).

If detecting an access to a not yet initialized *entity* is not possible, then runtime evaluation is performed as follows:

- Default value is produced if the type of an entity has a default value;
- Otherwise, `NullPointerError` is thrown.

15.13 Compatibility Features

Some features are added to ArkTS in order to support smooth TypeScript compatibility. Using these features while doing the ArkTS programming is not recommended in most cases.

15.13.1 Extended Conditional Expressions

ArkTS provides extended semantics for conditional expressions to ensure better TypeScript alignment. It affects the semantics of the following:

- *Ternary Conditional Expressions*;
- *Conditional-And Expression*;
- *Conditional-Or Expression*;

- *Logical Complement*;
- *while Statements and do Statements*;
- *for Statements*;
- *if Statements*.

Note

The extended semantics is to be deprecated in one of the future versions of ArkTS.

The extended semantics approach is based on the concept of *truthiness* that extends the boolean logic to operands of non-boolean types.

Depending on the kind of a valid expression's type, the value of the valid expression can be handled as `true` or `false` as described in the table below:

Value Type Kind	When `false`	When `true`	ArkTS Code Example to Check
`string`	empty string	non-empty string	`s.length == 0`
`boolean`	`false`	`true`	`x`
`char`	`c'\u0000'`	any value except `c'\u0000'`	`x`
`enum`	`enum` constant handled as `false`	`enum` constant handled as `true`	`x.valueOf()`
`number` (`double`/`float`)	`0` or `NaN`	any other number	`n != 0 && !isNaN(n)`
any integer type	`== 0`	`!= 0`	`i != 0`
`bigint`	`== 0n`	`!= 0n`	`i != 0n`
`null` or `undefined`	`always`	`never`	`x != null` or `x != undefined`
Union types	When value is `false` according to this column	When value is `true` according to this column	`x != null` or `x != undefined` for union types with nullish types
Any other nonNullish type	`never`	`always`	`new SomeType != null`

Extended semantics of *Conditional-And Expression* and *Conditional-Or Expression* affects the resultant type of expressions as follows:

- For *conditional-and* expression `A && B`:
  - If the value of `A` is known at compile time, then the type of an expression is the type of `B` when `A` is handled as `true`, and the type of `A` otherwise.
  - If the value of `A` is unknown at compile time, then the type of an expression is union `A | B`.
- For *conditional-or* expression `A || B`:
  - If the value of `A` is known at compile time, then the type of an expression is the type of `B` when `A` is handled as `false`, and the type of `A` otherwise.
  - If the value of `A` is unknown at compile time, then the type of an expression is union `A | B`.

The way this approach works in practice is represented in the example below. Any `nonzero` number is handled as `true`. The loop continues until it becomes `zero` that is handled as `false`:

```
console.log(typeof (false || 1) )
console.log(typeof (true || 1) )
for (let i = 10; i; i--) {
   console.log (i)
}
/* And the output will be
     int
     boolean
     10
     9
     8
     7
     6
     5
     4
     3
     2
     1
 */
```

CHAPTER

SIXTEEN

CONCURRENCY

ArkTS provides a wide variety of means for writing programs that efficiently utilize hardware resources by running certain pieces of code in parallel and asynchronously.

This chapter covers:

- the ArkTS execution model
- the semantics of *asynchronous*, *parallel* and *concurrent* code execution in ArkTS
- the language features and standard library API that provide the corresponding functionality

16.1 Execution model

A program in ArkTS defines one or more jobs that are executed by the runtime environment. A job is a piece of code that can be executed concurrently (that is, either asynchronously or in parallel with other jobs) and communicate its return value via the language provided mechanism.

Given that the target platform allows for parallel code execution, a worker thread is an abstraction over platform provided unit of parallelism. Typically, it will map 1:1 with OS threads. That means:

- every job is hosted by a worker thread with only one job per worker thread being executed at once
- if two or more jobs run on different worker threads then their code is able to run in parallel (this execution mode will be referred to as *parallel execution*)
- if several jobs share the same worker thread then their code can never run in parallel (this execution mode will be referred to as *asynchronous execution*).

A job's body has the starting point (the beginning of the corresponding piece of code) and the completion point (the end of the corresponding piece of code). A job's body can (but is not obliged to) map 1:1 to such language entities as functions, methods or lambdas.

A job can have zero or more suspension points. Execution of a job can be paused at a suspension point and resumed at a later moment in time. Once suspended, a job allows another job to be executed on the same worker thread.

Any ArkTS program implicitly defines one **main** job, which corresponds to the *Program Entry Point*. The execution starts from it, and its completion initiates the program termination sequence.

The exact language features and standard library APIs that are used for defining jobs and their respective suspension points are described in the subsequent sections.

The program memory is shared between all jobs , which allows for efficient data sharing but implies that the developer should use the provided means of synchronization to avoid race conditions and guarantee thread safety.

16.2 Overview of concurrency features

ArkTS allows for both asynchronous and parallel programming and provides the following:

- *Asynchronous execution* primitives: `async` / `await` / `Promise`;
- *Parallel Execution* API: `EAWorker API` / `Taskpool API`, including the structured concurrency support;
- *Synchronization* API: locks API, atomics API and other means of synchronization;

The *API details and restrictions* section provides the detailed API description and the restrictions on its usage.

16.3 Asynchronous execution

The *asynchronous execution* capability addresses the situation when developer's code regularly needs to wait for external (e.g network, timers or user input) or internal (e.g. status updates from a job that is running on another worker thread) events. For such cases, ArkTS provides a way to suspend execution of a job, mark the job as blocked on a wait for certain event and resume its execution later, once the event happens.

The ArkTS features that provide the asynchronous execution support are:

- the `async` and `await` keywords that mark suspendable (`asynchronous`) functions and suspension points inside such functions, respectively
- the `Promise` class in the Standard Library, which is an abstraction of the unfinished computation result that will get its value at some time in future.

16.3.1 Asynchronous Functions

An *asynchronous function* is a function that can define suspension points inside its body. A non-asynchronous function can not have suspension points.

> **Note**
>
> A *suspension point* is the point in the function code where function execution can be paused (so the function becomes *suspended*) and, at some time in future, *resumed*. The suspension implies that control is transferred elsewhere, but all the local function state (e.g. the argument and local variable values) is preserved until the resumption.

Asynchronous functions should be marked with the `async` modifier and return an instance of the generic *Promise class* class, which wraps the return value. The `async` modifier is not a part of the function type: a `non-async` function that returns `Promise` and an `async` function with the same return type and arguments are no different from the type system perspective. The function that serves as the *Program Entry Point* can be asynchronous, too.

Execution of an `async` function can be paused at suspension points, which are defined with the *await expression*. If suspension happens, the `async` function immediately returns a *pending* `Promise` instance. In case of the first (by the control flow) suspension point, the control is transferred to the caller of the asynchronous function, and the runtime environment creates a new suspendable job that corresponds to the suspended function. Eventually, in accordance with the *Scheduling rules*, the `async` function resumes its execution from the suspension point where the execution

was paused. In case of second and further suspension points, after the suspension happens, the runtime environment schedules another job for execution. The job to be scheduled is chosen in accordance with the *Scheduling rules*.

Execution of an asynchronous function completes either by a normal return or by throwing an error. In both cases, the resulting value or error is wrapped into a `Promise` class instance, *resolving* or *rejecting* it respectively (see *Promise class* for details).

An asynchronous function with the return type `Promise<T>` can explicitly return a `Promise<T>` instance (in this case, the returned value is returned "as is") or a value of type `T`, which is then automatically boxed in an instance of `Promise<T>`. Both options are allowed to be the `expression` of the `return` statement inside the `async` function body (see *return Statements* and *Return Type Inference*). `T` here is a subtype of *Type Any*. If `T` has `void` or `undefined` type (see *Type void or undefined*) then, like in non-asynchronous functions, an argumentless `return` statement is allowed.

Note

Summarizing: an asynchronous function with one or more suspension points defines a new suspendable job in an ArkTS program, which starts from the first suspension and ends with the asynchronous function completion. An asynchronous function with zero suspension points does not define any additional jobs.

A compile-time error occurs if:

- `async` function is called in a static initialization;
- `async` function has an `abstract` or a `native` modifier;
- return type of an `async` function is other than `Promise`.
- a `non-async` function defines any suspension points

16.3.2 Asynchronous Lambdas

A lambda can have the `async` modifier (see *Lambda Expressions* and *Trailing Lambdas*).

With regard to concurrency, `async` lambdas have the same semantics and follow the same rules as *Asynchronous Functions*.

16.3.3 Asynchronous Methods

A static or instance class method can have the `async` modifier (see *Method Declarations*).

With regard to concurrency, `async` methods have the same semantics and follow the same rules as *Asynchronous Functions*.

16.3.4 `await` expression

An *await expression* defines a suspension point within the asynchronous function body. The syntax of the *await expression* is as follows:

```
awaitExpression:
    'await' expression
    ;
```

The `expression` argument here can be of any type (*Type Any*). The type of an *await expression* is `Awaited<type(expression)>` (see *Awaited Utility Type*), but the value and the semantics of the *await expression* depend on the type of its `expression` argument.

If the type of an `expression` is a subtype of *Promise class*, then:

- Execution of the enclosing asynchronous function is paused until the awaited `Promise` instance is *resolved* or *rejected*;
- If the awaited `Promise` instance is *resolved*, then the value with which the `Promise` is resolved becomes the value of the *await expression*;
- If the awaited `Promise` instance is *rejected*, then the *await expression* throws the error with which the `Promise` instance is rejected;

If the `expression` type is not a subtype of *Promise class*, then the *await expression* returns the value of the `expression` argument, and the enclosing asynchronous function is not suspended:

```
class SomeClass {
  method(): SomeClass | undefined { /* body */ }
  async asyncMethod(): Promise<string> { /* body */ }
}

async function g(): Promise<Object> { /* returns Promise */ }

async function f() { // await is allowed in async context only
  // ...

  // v1 is Awaited<Promise<Object>>, which is effectively Object
  // g returns Promise, hence f can be suspended potentially
  let v1 = await g()

  // v2 is Awaited<Int>, which is effectively Int
  // await argument is an Int, hence no suspension occurs
  let v2 = await new Int(5)

  // v3 is Awaited<Promise<string> | undefined>, which is (string | undefined)
  // - if method() returned an object: suspension can occur, v3 is the await result
  // - if method() returned undefined: no suspension, v3 is undefined
  let v3 = await (new SomeClass).method()?.asyncMethod()

  // ...
}
```

Under certain circumstances, the suspendable job that has been suspended on `await` can be moved to another worker thread upon resumption, i.e. rescheduled (see *Scheduling rules*).

A compile-time error occurs if `await` is used outside of an asynchronous function, method or lambda body.

16.3.5 `Promise` class

The `Promise` class represents a value that is to be defined at some time in future, thus allowing for referencing a result of an unfinished calculation or an incomplete task. All kinds of jobs in ArkTS use promises to communicate their results to the client code.

A `Promise` instance can have the following states:

- *pending*: this state means that the resulting value is not yet known;
- *resolved*: this state means that the `Promise` has been *fulfilled* and its value has been defined;
- *rejected*: this state means that the associated calculation completed abnormally, so the `Promise` instance contains the error instance that describes the reason of abnormal completion;

The only way to get the value that was used to resolve or reject the `Promise` is to apply the *await expression* to the `Promise` instance.

> **Note**
>
> The semantics of `Promise` is similar to the semantics of `Promise` in JavaScript/TypeScript if it was returned by an asynchronous function on the **main** worker thread or created manually on the **main** worker thread. It is to be defined if such statements should reside in ArkTS specification.

In general, the `Promise` instances are safe to be accessed concurrently. The exceptions for this rule and the detailed API is described in the *API details and restrictions* section.

16.4 Parallel Execution

The *parallel execution* capability addresses the situation when developer's code executes either CPU-intensive tasks that can take advantage of utilizing multiple CPU cores or long tasks that can take advantage of running in a separate OS thread of execution to avoid blocking.

For such cases, ArkTS provides a standard library level API that allows for running code in parallel at function/method granularity (that is, for defining jobs that can run on different worker threads), with the ability to specify dependencies between jobs and balance the load across the available CPU cores and/or OS threads.

This capability is orthogonal to the *Asynchronous execution* capability, i.e. asynchronous functions can also be run in parallel, and this in general does not affect the way they are suspended/resumed. The only difference is that under certain conditions a suspendable job can change its worker thread upon resumption (see *Scheduling rules*).

16.4.1 Parallel execution API

ArkTS standard library provides the following sets of API for parallel execution:

- *EAWorker API*: the API that allows for creation of worker threads that are used exclusively by a job and its children;
- *Taskpool API*: the framework that offers structured concurrency capabilities: task grouping, dependencies and cancellation

16.4.2 EAWorker API

The EAWorker API is designed for the use case when developer's code requires a dedicated worker thread to run on (for example, such use case is relevant for UI frameworks).

This API creates a worker thread for the *exclusive* use of an initial job. That means, the initial job and all the jobs spawned by it will stay on this newly created worker thread, and no other job can be scheduled to this worker thread.

Please refer to the standard library documentation to find out more information.

16.4.3 Taskpool API

The `taskpool` is the structured concurrency framework. It allows to create new jobs, specify dependencies between them, cancel the spawned jobs, combine them in groups and choose a complex execution order.

> **Note**
>
> The suspendable jobs created by the taskpool API can not be rescheduled to another worker thread upon resumption.

Please refer to the standard library documentation to find out more information.

16.5 Synchronization

The synchronization mechanisms that exist in ArkTS and its standard library address the need for imposing certain order on the execution of the code that belongs to the jobs being executed asynchronously or in parallel. Such need originates from the two facts:

- firstly, all the data in ArkTS are by default shared between all jobs on all worker threads, which may cause data races in case when the same data is accessed concurrently;
- secondly, certain code sequences expect the data they operate on to be accessed exclusively by their job. If this job is a suspendable job and it suspends its execution inbetween of such code sequence, then another job can violate the expected exclusivity even in case when it runs on the same worker thread.

The means of synchronization that ArkTS provides are:

- *Asynchronous lock*: the "fused" asynchronous locking API, which allows the provided callback to safe operate on some data;
- *Asynchronous mutex*, *Asynchronous read/write lock*, *Asynchronous condition variable*: the "decoupled" asynchronous locking API, which provides the asynchronous version of traditional decoupled `lock()` / `unlock()` operations and also an asynchronous condition variable equivalent;
- *Atomic operations*: the atomic operations support, which also allows for building efficient lock-free structures
- *Additional entities and other notes*: a variety of other supplemental standard library classes and methods

16.5.1 Asynchronous lock

The asynchronous lock (`AsyncLock` class) allows to protect some shared data, for example, a part of object state, from concurrent access. It is designed for the use cases when the code sequence that accesses the protected state can be conveniently isolated as a distinct function object (function, method or lambda).

```
// a shared (e.g. global) data that we would like to protect
class SharedState {
  value: string = "nothing"
}
let whatIsInTheBag: SharedState = new SharedState

// a function that reads and modifies the shared data
function checkAndModify(data: SharedState, expected: string, updated: string) {
  if (data.value != expected) {
    // data race!
    console.log("race!")
  }
  data.value = updated
}

// a suspension point emulator
async function delay() {
  return new Promise<void>((res, rej) => {
    setTimeout(res, 1, undefined)
  })
}

// create a lock somewhere, for example as a global variable
let lock = new AsyncLock()

async function f() {
 // request an operation under the specified lock
 let p1 = lock.lockAsync<void>(async () => {
     // once the request can be satisfied, this lambda will run on the same
     // worker thread with the lock acquired

     // execute a modification sequence on the protected data:
     // nothing -> paraglider -> nothing

     // nothing -> paraglider
     checkAndModify(whatIsInTheBag, "nothing", "paraglider")

     // a sample suspension point that simulates a real life situation when
     // the modification sequence gets paused and another async function on
     // the same worker thread gets control
     await delay()

     // continue with the modification
     // paraglider -> nothing
     checkAndModify(whatIsInTheBag, "paraglider", "nothing")
```

(continues on next page)

(continued from previous page)

```
}, AsyncLockMode.EXCLUSIVE)

// request another operation under the same lock: it will be executed not
// earlier than the lock can be acquired
let p2 = lock.lockAsync<void>(async () => {
    // another asynchronous modification sequence that suspends inbetween:
    // nothing -> apple -> nothing

    checkAndModify(whatIsInTheBag, "nothing", "apple")
    await delay()
    checkAndModify(whatIsInTheBag, "apple", "nothing")
}) // AsyncLockMode.EXCLUSIVE is the default, so it can be skipped

// wait for both operations to complete
await p1
await p2

// Since both sequences have suspension points within them, they were
// executed in an interleaved manner if there would no locks, which would
// cause a data race and thus an error.
// However, the lock that we used for synchronization prevents this
// situation and each modification sequence executes as a critical section,
// which leads to the correct result.
}

function main() {
  f()
}
```

A developer can request one of two levels of access exclusivity to the given `AsyncLock`: exclusive or shared. The difference is as follows:

- if an exclusive access is requested (default behaviour), then no other request for callback execution under the same `AsyncLock` instance will be satisfied until the requester's callback is finished;
- if a shared access is requested then any other request for the callback execution under this lock can be concurrently satisfied, but all requests that demand exclusive access will wait their turn

  The callback execution under an `AsyncLock` can be safely requested concurrently both by the same and different jobs.

`AsyncLock` API provides a way to abort an existing request for callback execution and to query the status of the existing locks. Additionally it provides a way to limit the waiting time for an issued lock acquire request with a timeout and also gives hints about the potential deadlocks.

Please refer to the standard library documentation to find out more information.

16.5.2 Asynchronous mutex

The asynchronous mutex (`AsyncMutex`) allows to protect some shared data, for example, a part of object state, from concurrent access. It is designed for the following use cases:

- developer wants to use a condition variable (*Asynchronous condition variable*)
- the code sequence that accesses the protected state is hard to be conveniently isolated as a distinct function object (function, method or lambda), so the decoupled `lock()` and `unlock()` operations are required

```
// a shared (e.g. global) data that we would like to protect
class SharedState {
  value: string = "nothing"
}
let whatIsInTheBag: SharedState = new SharedState

// a function that reads and modifies the shared data
function checkAndModify(data: SharedState, expected: string, updated: string) {
  if (data.value != expected) {
    // data race!
    console.log("race!")
  }
  data.value = updated
}

// a suspension point emulator
async function delay() {
  return new Promise<void>((res, rej) => {
    setTimeout(res, 1, undefined)
  })
}

// create a lock somewhere, for example as a global variable
let lock = new AsyncMutex()

async function f() {
 // here we execute a modification sequence on the protected data under the
 // lock: nothing -> paraglider -> nothing

 // the await is mandatory! the promise returned by the lock() method will
 // get resolved not earlier than the lock is successfullly acquired
 await lock.lock();
 // the code between lock() and unlock() acts like a critical section:
 // no other job is able to acquire the "lock" till the "unlock()" is called

 // nothing -> paraglider
 checkAndModify(whatIsInTheBag, "nothing", "paraglider")

 // a sample suspension point that simulates a real life situation when
 // the modification sequence gets paused and another async function on
 // the same worker thread gets control
 await delay()

 // continue with the modification
 // paraglider -> nothing
 checkAndModify(whatIsInTheBag, "paraglider", "nothing")

 // end of the critical section
 lock.unlock()
}

async function g() {
 // another asynchronous modification sequence that suspends inbetween:
```

(continues on next page)

(continued from previous page)

```
 // nothing -> apple -> nothing
 await lock.lock()
 // start of the critical section

 checkAndModify(whatIsInTheBag, "nothing", "apple")
 await delay()
 checkAndModify(whatIsInTheBag, "apple", "nothing")

 // end of the critical section
 lock.unlock()
}

function main() {
 // Call both functions consequently without any waits.
 // Since both functions have suspension points within them, they were
 // executed in an interleaved manner if there would no locks, which would
 // cause a data race and thus an error.
 // However, the mutex we used for synchronization prevents this
 // situation and each modification sequence executes as a critical section,
 // which leads to the correct result.
  f()
  g()
}
```

The `AsyncMutex` can be safely used in both the jobs that run on the same worker thread and on different worker threads.

The avoidance of double locking (happens if the *lock()* method is called from the lock instance that is already acquired by the current job) and deadlocks is the developer's responsibility. Please refer to the standard library documentation to find out more information.

16.5.3 Asynchronous read/write lock

The asynchronous read/write lock (`AsyncRWLock`) allows to protect some shared data, for example, a part of object state, from concurrent access. It is designed for the use case when both of the following statements are true:

- the code sequence that accesses the protected state is hard to be conveniently isolated as a distinct function object (function, method or lambda), so the decoupled `lock()` and `unlock()` operations are required
- access to the shared state must be mutually exclusive between a group of entities that can safely access the data concurrently ("readers") and any other entity that requires exclusive access to the data ("writer"/"writers")

Please refer to the standard library documentation to find out more information.

16.5.4 Asynchronous condition variable

The asynchronous condition variable (`AsyncCondVar`) is designed for the use case when some shared data is used as a condition for a sequence of actions in one job and is concurrently modified in another job.

The use of `AsyncCondVar` requires *Asynchronous mutex*:

```
// create mutex and condition variable somewhere, e.g. in the global scope
let m = new AsyncMutex();
let cv = new AsyncCondVar();
// the shared data, which is used as a condition
let flag = false;
```

(continues on next page)

(continued from previous page)

```
async function f() {
 // the notification sequence (in job A):
 // lock the mutex
 await m.lock()
 // start of the critical section

 // update the condition
 flag = true
 // notify the waiter(s):
 // the API requires that the same mutex is to be provided here
 cv.notifyOne(m)

 // end of the critical section: unlock the mutex
 m.unlock()
}

async function g() {
 // the wait-and-react sequence (in job B):
 // lock the same mutex that is used for condition update and notification
 await m.lock() // await is mandatory!
 // start of the critical section

 // check the shared condition
 while (flag == false) {
   // start waiting for the condition to change:
   // the API requires that the same mutex is to be provided here
   // wait() unlocks "m" and returns the Promise that is going to be resolved
   // once some other job calls notifyOne()/notifyAll()
   await cv.wait(m) // await is mandatory!

   // at this point, "m" is locked again, and the notification has been received
 }
 // here the condition is satisfied, and the mutex is locked:
 // any dependent actions (e.g. some state update) can happen here, and they
 // effectively happen in the same critical section with the verification of
 // the shared condition value

 // end of the critical section: unlock the mutex
 m.unlock()
}

function main() {
  f()
  g()
}
```

Please refer to the standard library documentation to find out more information.

16.5.5 Atomic operations

ArkTS standard library provides a set of classes that support atomic operations. The intended use cases for them are lock free data structures and algorithms: from simple compare-and-swap and spinlocks to complex containers.

Please refer to the standard library documentation to find out more information.

16.5.6 Additional entities and other notes

The ArkTS standard library provides various additional classes and APIs that help developers to build safe and efficient concurrent programs. Such classes include:

- thread safe concurrent containers
- APIs that operate on worker thread-local data
- other helpers

Please refer to the standard library documentation to find out more information about them.

16.6 API details and restrictions

This section describes the noteworthy details and the notable restrictions for the concurrency APIs described in the previous sections.

16.6.1 Using the asynchronous API

In certain cases, a call to an `async` function requires awaiting its result, but the call site resides in the non-async function. In such cases, the caller function should be converted to an asynchronous one, and in some cases this chain of conversions has to be continued up to the program entry point. For this case, ArkTS supports the `async` entry point function (see *Program Entry Point*).

Note

Maybe, this section should be moved to the handbook.

16.6.2 Promise class API

There are some important restrictions that limit the correct usage of the `Promise` class.

A `Promise` class instance is safe to be awaited within a job on some worker thread while being resolved or rejected from another job on the same or different worker thread.

A `Promise` class allows to register callbacks that are to be called upon `Promise` resolution and/or rejection. This is done by calling the `.then()` / `.catch()` / `.finally()` methods of the `Promise` class. However, these methods have the following usage restrictions:

- the registered callback will be called as a separate job on the same worker thread where it was registered
- if multiple callbacks are registered from the jobs that reside on the same worker thread, the order of their execution matches the order of their registration
- if multiple callbacks are registered from the jobs that reside on different worker threads, the order of their execution is defined only within each worker thread, and no order is guaranteed between the resulting jobs that reside on different worker threads

Please refer to the standard library documentation to find out more information about the `Promise` methods.

16.6.3 Unhandled Rejected Promises

Note

The semantics of unhandled rejections will be revisited later, once the design of ArkTS concurrency subsystem is complete.

A rejected `Promise` is considered unhandled if, at certain time, there is no `await` waiting for it and there are no callbacks registered for it with the `.then()` / `.catch()` methods.

This moment of time is defined separately on each worker thread, hence the `Promise` instance is considered an *unhandled rejection* only within a context of some worker thread, while possibly being *handled* on other ones.

16.6.4 Error handling policy

In general, all errors thrown in an ArkTS program should either have an ability to be handled by the developer or considered uncaught, and initiate a program termination sequence. This applies to any job on any worker thread.

A job in an ArkTS program can complete abnormally, i.e., can throw an error. Since jobs communicate their return values using `Promise` class instances, in case of job's abnormal completion the corresponding promise gets rejected and the original error is not considered uncaught.

However, there can exist some cases when such rejection cannot be handled by the developer, for example:

- when the thrower job was created by the runtime environment, and no promise can be awaited or handled with a `.then()` / `.catch()` callback
- when the *main* job throws an error

In such cases, the original error thrown by the job will be considered uncaught.

16.7 Runtime implementation details

16.7.1 Scheduling rules

The runtime environment schedules the jobs that are defined by an ArkTS program in accordance with the following rules:

- Every job has a priority, which depends solely on its type. The list of types, from highest to lowest priority:
  1. suspendable jobs and `Promise` callbacks (`.then()`, etc.);
  2. Other jobs
- Within each worker thread, the jobs with higher priority are scheduled before jobs with lower priority;
- All jobs with the same priority are scheduled in the FIFO order;

> **Note**
>
> These rules are incomplete and will be updated.

CHAPTER

SEVENTEEN

EXPERIMENTAL FEATURES

This Chapter introduces the ArkTS features that are considered parts of the language, but have no counterpart in TypeScript, and are therefore not recommended to those who seek a single source code for TypeScript and ArkTS.

Some features introduced in this Chapter are still under discussion. They can be removed from the final version of the ArkTS specification. Once a feature introduced in this Chapter is approved and/or implemented, the corresponding section is moved to the body of the specification as appropriate.

The *array creation* feature introduced in *Resizable Array Creation Expressions* enables programmers to create objects of the array type at runtime by providing the following as arguments:

- Array size;
- One element to fill the array with, or a lambda to generate a set of elements to fill the array with.

This addition is useful to other array-related features of the language, such as array literals. This feature can also be used to create arrays of arrays.

Overloading functions, methods, or constructors is a practical and convenient way to write program actions that are similar in logic but different in implementation. ArkTS uses *Explicit Overload Declarations* as an innovative form of *managed overloading*.

Section *Native Functions and Methods* introduces practically important and useful mechanisms for the inclusion of components written in other languages into a program written in ArkTS.

Sections *Final Classes* and *Final Methods* discuss the well-known feature that in many OOP languages provides a way to restrict class inheritance and method overriding. Making a class *final* prohibits defining classes derived from it, whereas making a method *final* prevents it from overriding in derived classes.

Section *Adding Functionality to Existing Types* discusses the way to add new functionality to an already defined type.

17.1 Type `char`

Values of `char` type are 16-bit Unicode code units. Any Unicode code point can be encoded with one or two `char` values.

Type	Type's Set of Values
`char` (16-bits)	Symbols (code units) with codes from U+0000 to U+FFFF

Predefined constructors, methods, and constants for `char` type are parts of the ArkTS *Standard Library*.

Type `char` is a class type that is a part of the *Standard Library*. It means that type `char` is a subtype of `Object`, and that it can be used at any place where a class name is expected.

```
let a_char: char = c'a'
console.log (a_char)
// Output is: a
let o: Object = a_char // OK
```

17.1.1 char Literals

Char literal represents a 16-bit Unicode code unit that can be written as a single UTF-16 symbol or a single escape sequence preceded by the characters *single quote* (U+0027) and '*c*' (U+0063), and followed by a *single quote*.

The syntax of *character literal* is represented below:

```
CharLiteral:
    'c\'' SingleQuoteCharacter '\''
    ;

SingleQuoteCharacter:
    ~['\\\r\n]
    | '\\' EscapeSequence
    ;
```

The examples are presented below:

```
   c'a'
   c'\n'
   c'\x7F'
   c'\u0000'
```

If a literal cannot be represented by an unsigned 16-bit value, then a compile-time occurs:

```
   c'\u{FFFFF}' // Compile-time error
```

Char literals are of type `char`.

17.1.2 char Operations

Equality operators (see *Equality Expressions*) and relational operators *Relational Expressions*) can be used if:

- both operands are of `char` type; or
- one operand is of `char` type and other is of a numeric type (see *char Conversions for Relational and Equality Operands*);
- otherwise, a compile-time error occurs.

In first case, the operation is performed as an integer comparison of two unsigned 16-bit values. In the second case, the operation is performed as an integer comparison of the correspondent numeric type.

```
let c: char = c'A'
let c1 = new char
c1 = c'A'

// The following lines both print true as values are equal
console.log(c == c1)  // true
console.log(c === c1) // true

console.log(c == 0x41) // true

c1 = c'B'
console.log(c < c1)  // true
console.log(c < 0x41)  // false

console.log(c > 3.14)  // true
```

17.2 Fixed-Size Array Types

Fixed-size array type, written as `FixedArray<T>`, is the built-in type characterized by the following:

- Any instance of array type contains elements. The number of elements is known as *array length*, and can be accessed by using the property `length`.
- Array length is a non-negative integer number.
- Array length is set once at runtime and cannot be changed later.
- Array element is accessed by its index. *Index* is an integer number starting from *0* to *array length minus 1*.
- Accessing an element by its index is a constant-time operation.
- If passed to a non-ArkTS environment, an array is represented as a contiguous memory location.
- Type of each array element is assignable to the element's type specified in the array declaration (see *Assignability*).

Fixed-size arrays differ from *resizable arrays* as follows:

- Fixed-size array length is set once to achieve better performance;
- Fixed-size arrays preserve the element type during the *Type Erasure*;
- Fixed-size arrays have no methods defined;
- Fixed-size arrays have several constructors (see *Fixed-Size Array Creation*);
- Fixed-size arrays are not compatible with *resizable arrays*.

Incompatibility between a resizable array and a fixed-size array is represented by the example below:

```
function foo(a: FixedArray<number>, b: Array<number>) {
    a = b // Compile-time error
```

(continues on next page)

(continued from previous page)

```
    b = a // Compile-time error
}
```

17.2.1 Fixed-Size Array Creation

Fixed-size array can be created by using *Array Literal* or constructors defined for type `FixedArray<T>`, where `T` must be a type *preserved* by *Type Erasure*.

The use of an *array literal* to create an array is represented in following examples:

```
let a : FixedArray<number> = [1, 2, 3]
  /* create array with 3 elements of type number */
a[1] = 7 /* put 7 as the 2nd element of the array, index of this element is 1 */
let y = a[2] /* get the last element of array 'a' */
let count = a.length // get the number of array elements
y = a[3] // Will cause a runtime error - attempt to access non-existing array element
```

```
function foo<T>(v: T): FixedArray<T | number> {
  return [v] // Compile-time error, T | number is not preserved by type erasure
}
let arr: FixedArray<string | number> = foo("a")
```

The following constructor creates an instance of `FixedArray<T>` of the specified length, filled with a single value `elem`:

- `constructor(len: int, elem: T)`

```
let a = new FixedArray<string>(3, "a") // creates array ["a", "a", "a"]
```

17.3 Value Array Types

Value array type is the built-in type written as `ValueArray<T>` and characterized by the following:

- Any instance of array type contains elements of type `T`. `T` must be a *value type* (see *Value Types*).
- The number of elements is known as *array length*, and can be accessed by using the property `length`.
- Array length is a non-negative integer number.
- Array length is set once at runtime and cannot be changed later.
- Array element is accessed by its index. *Index* is an integer number starting from *0* to *array length minus 1*.
- Accessing an element by its index is a constant-time operation.
- If passed to a non-ArkTS environment, an array is represented as a contiguous memory location, filled by the primitive values (not references).
- Type of each array element is equal to the element's type specified in the array declaration.

- No subtyping relation holds between two `ValueArray` types, except where their type arguments are identical.

> **Note**
>
> - `ValueArray<T>` is not a generic type, despite using notation identical to generics.
> - Limitations imposed by `ValueArray` subtyping make it more performant compared to *Fixed-Size Array Types*.

Value array can be created by using *Array Literal* or constructors defined for type `ValueArray<T>` (see below).

The use of an *array literal* to create an array is represented in following examples:

```
let a : ValueArray<int> = [1, 2, 3]
  /* create array with 3 elements of type int */
a[1] = 7 /* put 7 as the 2nd element of the array, index of this element is 1 */
let y = a[2] /* get the last element of array 'a' */
let count = a.length // get the number of array elements
y = a[3] // runtime error, attempt to access non-existing array element
```

If `ValueArray` is used with non-value type argument, then a compile-time error occurs as follows:

```
let x: ValueArray<string> = ["aa"]   // Compile-time error
type A = ValueArray<int | undefined> // Compile-time error
```

The following constructor creates an instance of `ValueArray<T>` of the specified length, filled with a single value `elem`:

- `constructor(len: int, elem: T)`

```
let a = new ValueArray<double>(3, 7.) // creates array [7., 7., 7.]
```

17.4 Resizable Array Creation Expressions

Array creation expression creates new objects that are instances of *resizable arrays* (see *Resizable Array Types*). An array instance can be created alternatively by using *Array Literal*.

The syntax of *array creation expression* is presented below:

```
newArrayInstance:
    'new' arrayElementType dimensionExpression '(' arrayElement ')'
    ;

arrayElementType:
    typeReference
    | '(' type ')'
    ;

dimensionExpression:
    '[' expression ']'
```

(continues on next page)

(continued from previous page)

```
    ;

arrayElement:
  expression
;
```

```
let x = new number[3] (7) // create array instance: [7, 7, 7]
```

Array creation expression creates an object that is a new array with the elements of the type specified by `arrayElementType`.

The type of the *dimension expression* must be assignable (see *Assignability*) to an `int` type. Otherwise, a compile-time error occurs.

A compile-time error occurs if the *dimension expression* is a constant expression that is evaluated to a negative integer value at compile time.

Type of `arrayElement expression` must be be assignable (see *Assignability*) to `arrayElementType`. Otherwise, a compile-time error occurs.

```
let x = new number[-3] (0) // Compile-time error

let y = new number[3.14] (0) // Compile-time error

function foo (length: int) {
   let y = new number[length] (0)  // runtime error
}
foo (-3)
```

A compile-time error occurs if `arrayElementType` is a type parameter:

```
class A<T> {
   foo(element: T) {
      new T[2] (element) // Compile-time error, 'T' is a type parameter
   }
}
```

The creation of an array with a known number of elements is presented below:

```
class A {
  constructor (x: number) {}
}

let array_size = 5

let array = new A[array_size] (new A(1))
   /* Create array of 'array_size' elements and all of them will have
      initial value equal to an object created by new A expression */
```

The creation of exotic arrays with different kinds of element types is presented below:

```
let array_of_union = new (Object|undefined) [5] (undefined) // filled with undefined

type Functor = () => void
```

(continues on next page)

(continued from previous page)

```
let array_of_functor = new Functor[5] ( (): void => {}) // filled with lambda

type Arr = number []
let array_of_array = new Arr [5] ( [3.14] ) // filled with array literal
```

17.4.1 Runtime Evaluation of Array Creation Expressions

The evaluation of an array creation expression at runtime is performed as follows:

1. The dimension expression is evaluated. If the dimension expression evaluation completes abruptly, then *array creation expression* also does so.
2. The value of dimension expression is checked. If its value is less than zero, then `NegativeArraySizeError` is thrown.
3. Space for the new array is allocated. If the available space is not sufficient to allocate the array, then `OutOfMemoryError` is thrown, and the evaluation of the array creation expression completes abruptly.
4. Then, a one-dimensional array is created. Each element of this array is initialized either with the value passed or by calls to the lambda generating a set of values.

17.5 Indexable Types

If a class or an interface declares one or two functions with names `$_get` and `$_set`, and signatures *(index: Type1): Type2* and *(index: Type1, value: Type2)* respectively, then an indexing expression (see *Indexing Expressions*) can be applied to variables of such types:

```
class SomeClass {
   $_get (index: number): SomeClass { return this }
   $_set (index: number, value: SomeClass) { }
}
let x = new SomeClass
x = x[1] // This notation implies a call: x = x.$_get (1)
x[1] = x // This notation implies a call: x.$_set (1, x)
```

If only one function is present, then only the appropriate form of indexing expression (see *Indexing Expressions*) is available:

```
class ClassWithGet {
   $_get (index: number): ClassWithGet { return this }
}
let getClass = new ClassWithGet
getClass = getClass[0]
getClass[0] = getClass // Error - no $_set function available

class ClassWithSet {
```

(continues on next page)

(continued from previous page)

```
    $_set (index: number, value: ClassWithSet) { }
}
let setClass = new ClassWithSet
setClass = setClass[0] // Error - no $_get function available
setClass[0] = setClass
```

Type `string` can be used as a type of the index parameter:

```
class SomeClass {
    $_get (index: string): SomeClass { return this }
    $_set (index: string, value: SomeClass) { }
}
let x = new SomeClass
x = x["index string"]
    // This notation implies a call: x = x.$_get ("index string")
x["index string"] = x
    // This notation implies a call: x.$_set ("index string", x)
```

Functions `$_get` and `$_set` are ordinary functions with compiler-known signatures. The functions can be used like any other function. The functions can be abstract, or defined in an interface and implemented later. The functions can be overridden and provide a dynamic dispatch for the indexing expression evaluation (see *Indexing Expressions*). The functions can be used in generic classes and interfaces for better flexibility. A compile-time error occurs if these functions are marked as `async`.

```
interface ReadonlyIndexable<K, V> {
    $_get (index: K): V
}

interface Indexable<K, V> extends ReadonlyIndexable<K, V> {
    $_set (index: K, value: V)
}

class IndexableByNumber<V> implements Indexable<number, V> {
    private data: V[] = []
    $_get (index: number): V { return this.data [index] }
    $_set (index: number, value: V) { this.data[index] = value }
}

class IndexableByString<V> implements Indexable<string, V> {
    private data = new Map<string, V>
    $_get (index: string): V { return this.data [index] }
    $_set (index: string, value: V) { this.data[index] = value }
}

class BadClass extends IndexableByNumber<boolean> {
    override $_set (index: number, value: boolean) { index / 0 }
}

let x: IndexableByNumber<boolean> = new BadClass
x[42] = true // This will be dispatched at runtime to the overridden
    // version of the $_set method
x.$_get (15)  // $_get and $_set can be called as ordinary
    // methods
```

17.6 Iterable Types

A class or an interface is *iterable* if it implements the interface `Iterable` defined in the *Standard Library*, and thus has an accessible parameterless method with the name `$_iterator` and a return type that is a subtype (see *Subtyping*) of type `Iterator` as defined in the *Standard Library*. It guarantees that an object returned by the `$_iterator` method is of the type which implements `Iterator`, and thus allows traversing an object of the *iterable* type.

A union of iterable types is also *iterable*. It means that instances of such types can be used in `for-of` statements (see *for-of Statements*).

Array (see *Array Types*) and string (see *Type string*) types are iterable.

An *iterable* class `C` is represented in the example below:

```
class C implements Iterable<string> {
  data: string[] = ['a', 'b', 'c']
  $_iterator() { // Return type is inferred from the method body
    return new CIterator(this)
  }
}

class CIterator implements Iterator<string> {
  index = 0
  base: C
  constructor (base: C) {
    this.base = base
  }
  next(): IteratorResult<string> {
    return {
      done: this.index >= this.base.data.length,
      value: this.index >= this.base.data.length ? "" : this.base.data[this.index++]
    }
  }
}

let c = new C()
for (let x of c) {
      console.log(x)
}
```

In the example above, class `C` method `$_iterator` returns `CIterator<string>` that implements `Iterator<string>`. If executed, this code prints out the following:

```
"a"
"b"
"c"
```

The method `$_iterator` is an ordinary method with a compiler-known signature. This method can be used like any other method. It can be abstract or defined in an interface to be implemented later. A compile-time error occurs if this method is marked as `async`.

Note

To support the code compatible with TypeScript, the name of the method `$_iterator` can be written as `[Symbol.iterator]`. In this case, the class `C` from the example above looks as follows:

```
class C implements Iterable<string>  {
  data: string[] = ['a', 'b', 'c'];
  [Symbol.iterator]() {
    return new CIterator(this)
  }
}
```

The use of the name `[Symbol.iterator]` is considered deprecated. It can be removed in the future versions of the language.

17.7 Callable Types

A type is *callable* if the name of the type can be used in a call expression. A call expression that uses the name of a type is called a *type call expression*. Only class type can be callable. To make a type callable, a static method either with the name `$_invoke` or with the name `$_instantiate` must be defined:

```
class C {
    static $_invoke() { console.log("invoked") }
}
C() // prints: invoked
C.$_invoke() // also prints: invoked
```

In the above example, `C()` is a *type call expression*. It is the short form of the normal method call `C.$_invoke()`. Using an explicit call is always valid for the methods `$_invoke` and `$_instantiate`.

A class can define either the method `$_invoke()` or the method `$_instantiate` but not both. Otherwise, a compile-time error occurs. However, a class can define several implementations of the methods `$_invoke` or `$_instantiate` with different signatures:

```
// Compile-time error, both $_invoke and $_instantiate defined
class A {
    static $_invoke(i: int): int { return i; }
    static $_instantiate(factory: () => A): A { return factory(); }
}

// OK, two $_invoke with different signatures
class B {
    static $_invoke(p: int): int { return p; }
    static $_invoke(): string { return "hello"; }
}
```

Static methods have no access to type parameters of generic in ArkTS. It means that the method `$_instantiate` cannot be declared for a generic type. The method `$_invoke` can be declared, but the *type call expression* or explicit call of `$_invoke()` must not use a type parameter.

Note

Only a constructor—not the methods `$_invoke` or `$_instantiate`—is called in a *new expression*:

```
class C {
    static $_invoke() { console.log("invoked") }
    constructor() { console.log("constructed") }
}
let x = new C() // constructor is called
```

The methods `$_invoke` and `$_instantiate` are similar but have differences as discussed below.

17.7.1 Callable Types with `$_invoke` Method

The static method `$_invoke` can have an arbitrary signature. The method is either called implicitly in a *type call expression*, or called explicitly. The class can have several `$_invoke` methods with different signatures. If the signature has parameters, then the call must contain corresponding arguments.

```
class Add {
    static $_invoke(a: number, b: number): number {
        return a + b
    }
    static $_invoke(a: string, b: string): string {
        return a + b
    }
}
console.log(Add(2, 2)) // prints: 4
console.log(Add.$_invoke(2, 2)) // prints: 4
console.log(Add("Number ", "one")) // prints "Number one"
```

A class can declare an instance method `$_invoke` but the method does not make the class *callable*.

17.7.2 Callable Types with `$_instantiate` Method

The static method `$_instantiate` can have an arbitrary signature by itself. If it is to be used in a *type call expression*, then its first parameter must be a *factory* defined as a parameterless function type returning the class type in which the method `$_instantiate` is declared. The method can have or not have other parameters which can be arbitrary. The return type of the method `$_instantiate` is typically the same as the return type of the factory, but can be arbitrary instead. A class can contain several static `$_instantiate` methods with different sets of parameters. If a class declares two `$_instantiate` methods that have the same parameter set but different return types, then a compile-time error occurs.

In a *type call expression*, the argument corresponding to the `factory` parameter is passed implicitly:

```
class C {
    // #1, parameterless
```

(continues on next page)

(continued from previous page)

```
    static $_instantiate(factory: () => C): C {
        return factory()
    }

    // #2. As #1, but with another return type
    // If uncommented, then a compile-time error occurs
    // static $_instantiate(factory: () => C): int {
    //     return 1
    // }

    // #3, with string parameter
    static $_instantiate(factory: () => C, s: string): string {
        return "hello " + s
    }
}

let x = C() // #1 called, factory is passed implicitly

// Explicit call of #1 requires explicit 'factory':
let y = C.$_instantiate(() => { return new C()})

let s: string = C("world") // #3 called, factory is passed implicitly
```

If the method `$_instantiate` has additional parameters, then the call must contain corresponding arguments:

```
class C {
    name = ""
    static $_instantiate(factory: () => C, name: string): C {
        let x = factory()
        x.name = name
        return x
    }
}
let x = C("Bob") // factory is passed implicitly
```

A compile-time error occurs in a *type call expression* with type `T`, if:

- `T` has neither method `$_invoke` nor method `$_instantiate`; or
- `T` has the method `$_instantiate` but its first parameter is not a `factory`.

```
class C {
    static $_instantiate(factory: string): C {
        return factory()
    }
}
let x = C() // Compile-time error, wrong '$_instantiate' 1st parameter
```

Where the method `$_instantiate` is used implicitly in the *type call expression*:

- If the method `$_instantiate` does not declare `factory` as an *optional parameter*, then a `factory` implementation is generated automatically.
- If the method `$_instantiate` declares `factory` as an *optional parameter* (see *Optional Parameters*), then the default implementation is used for `factory`.

```
class A {
    static $_instantiate(
        factory: () => A): A { return factory() }
}
class B {
    static $_instantiate(
        factory: () => B = () =>{
            console.log("default factory");
            return new B; } ): B
        { return factory() }
}

A() // Automatically generated factory is used
B() // Default implementation is used for factory
```

A *type call expression* passes no arguments to the `factory` function, and the latter uses a parameterless class constructor. If a class has no parameterless class constructor, then a compile-time error occurs:

```
class A {
    constructor(p: int) {}
    static $_instantiate(factory: () => A): A { return factory() }
}

A() // Compile-time error, no parameterless constructor
```

Note

Calling the method `$_instantiate` explicitly with such a class, or supplying the default implementation for the factory that uses a constructor with parameters, is still possible though useless:

```
class A {
    constructor(p: int) {}
    static $_instantiate(
        factory: () => A = () => { return new A(1); }
        ): A { return factory() }
}

A() // OK, default is used for the optional factory
A.$_instantiate(() => { return new A(1); }) // OK, explicit call
```

A class can declare an instance method `$_instantiate` but the method does not make the class *callable*.

17.8 Statements

17.8.1 For-of Explicit Type Annotation

An explicit type annotation is allowed for a *ForVariable* (see *for-of Statements*):

```
// explicit type is used for a new variable,
let x: string[] = ["aaa", "bbb", "ccc"]
for (let str: string of x) {
  console.log(str)
}
```

Type of elements in a `for-of` expression must be assignable (see *Assignability*) to the type of the variable. Otherwise, a compile-time error occurs.

17.9 Explicit Overload Declarations

ArkTS supports conventional overloading for functions, methods, and constructors (i.e. implicit overloading of same-name entities), and an innovative form of explicit overload declarations that allows a developer to specify a set of overloaded entities explicitly and to control the overload resolution process.

Regardless of implicit or explicit overloading being used, the actual entity to be called is determined at compile time. As a result, *overloading* is related to *compile-time polymorphism by name*. The semantic details are discussed in *Overloading*.

An *explicit overload declaration* can be used for:

- Functions (see *Explicit Function Overload*), including functions in namespaces;
- Class or interface methods (see *Explicit Class Method Overload* and *Explicit Interface Method Overload*).

An *overload declaration* starts with the keyword `overload` and declares an *ordered overload set*. The exact syntax of the declaration is presented in the appropriate subsections.

The use of an explicit overload declaration is represented in the example below:

```
function max2(a: number, b: number): number {
    return  a > b ? a : b
}
function maxN(...a: number[]): number {
    // return max element
}

// declare 'max' as an ordered set of functions max2 and maxN
overload max { max2, maxN }

max(1, 2)     // max2 is called
max(3, 2, 4)  // maxN is called
max("a", "b") // Compile-time error, no function to call
```

The semantics of an entity included into an *overload set* does not change. Such entities follow the ordinary accessibility rules, and can be called explicitly as follows:

```
maxN(1, 2) // maxN is explicitly called
max2(2, 3) // max2 is explicitly called
```

When calling an *explicit overload*, entities from an *overload set* are checked in the listed order, and the first entity with an appropriate signature is called (see *Overload Resolution* for detail). A compile-time error occurs if no entity with an appropriate signature is available:

```
max(1)    // maxN is called
max(1, 2) // max2 is called, as is the first appropriate in the set

max("a", "b") // Compile-time error, no function to call
```

Each identifier in an *explicit overload declaration* must denote exactly one accessible entity.

If an identifier denotes an implicitly overloaded name, then a compile-time error occurs.

An *explicit overload declaration* never expands an implicitly overloaded name into several entities.

An overloaded entity in an *explicit overload declaration* can be *generic* (see *Generics*).

If type arguments are provided explicitly in a call of an overloaded entity (see *Explicit Generic Instantiations*), then only the entities that have the number of type arguments compatible with the number of mandatory and optional type parameters (i.e., entities with optional type parameters are the entities that have type parameter default) are considered during *Overload Resolution*:

```
// Resolution with explicit type arguments
function one<T>() { console.log("one") }
function two<T, U=string>() { console.log("two") }
function three<T, U=string, V=number>() { console.log("three") }

overload numbers { one, two, three }

numbers<string, number, int>() // Only 'three` considered

numbers<string, number>() // 'two' and 'three; considered as both
                          // allow 2 type arguments

numbers<int>()  // 'one', 'two' and 'three; considered as all
                // allow 1 type argument
```

If *type arguments* are not provided explicitly (see *Implicit Generic Instantiations*), then consideration is given to all entities as represented in the example below:

```
function foo1(s: string) {}
function foo2<T>(x: T) {}

overload foo { foo1, foo2 }

foo("aa")   // foo1 is called
foo(1) // foo2 is called, implicit generic instantiation
foo<string>("aa") // foo2 is called
```

An entity can be listed in several *explicit overload declarations*:

```
function max2i(a: int, b: int): int {
    return  a > b ? a : b
}
function maxNi(...a: int[]): int {
    // return max element
```

(continues on next page)

(continued from previous page)

```
}
function maxN(...a: number[]): number {
    // return max element
}

overload maxi { max2i, maxNi }
overload max { max2i, maxNi, maxN }
```

17.9.1 Explicit Function Overload

Explicit function overload allows declaring a name for a set of functions (see *Function Declarations*). The syntax is presented below:

```
explicitFunctionOverload:
    'overload' identifier overloadList
    ;
overloadList:
    '{' identifier (',' identifier)* ','? '}'
    ;
```

A compile-time error occurs if an *identifier* in the list:

- refers to no accessible function;
- refers to an implicitly overloaded function name; or
- refers to a non-function entity.

All overloaded functions must be in the same module or namespace scope (see *Scopes*). Otherwise, a compile-time error occurs. The erroneous overload declarations are represented in the example below:

```
import {foo1} from "something"

function foo2() {}
overload foo {foo1, foo2} // Compile-time error

namespace N {
    export function fooN() {}
    namespace M {
        export function fooM() {}
    }
    overload goo {M.fooM, fooN} // Compile-time error
}
overload bar {foo2, N.fooN} // Compile-time error
```

A name of an *explicit function overload* can be the same as the name of a function in the same scope only if that name denotes exactly one accessible function and that function is used in the overloaded list. If that name denotes an implicitly overloaded function set, then a compile-time error occurs. This situation is represented in the following example:

```
function foo(n: number): number {/*body1*/}
function fooString(s: string): string {/*body2*/}

overload foo {foo, fooString} // valid overload

foo(1)    // function 'foo' is called
foo("aa") // function 'fooString' is called

function bar(): void {}

// Invalid overload, as 'bar' does not appear in the list:
overload bar {foo, fooString} // Compile-time error

function baz(n: number): number {/*body1*/}
function baz(s: string): string {/*body2*/}

// Invalid overload, as 'baz' denotes an implicitly overloaded function set:
overload baz {baz, fooString} // Compile-time error

let name: string = "abc"

// Invalid overload, as 'name' refers to a variable:
overload name {foo, fooString, name} // Compile-time error
```

Using the name of a function as the name of an explicit overload causes no ambiguity for it is considered at the call site only, i.e., a name of an *explicit overload* is **not** considered in the following situations:

- List of the overloaded entities;
- *Function Reference*.

```
function foo(n: number): number {/*body1*/}
function fooString(s: string): string {/*body2*/}
overload foo {foo, fooString}

let func1 = foo // 'foo' refers to function, not to explicit overload
```

A compile-time error occurs, if an *explicit overload* is exported but an overloaded function is not:

```
export function foo1(p: string) {}
function foo2(p: number) {}
export overload foo { foo1, foo2 } // Compile-time error, 'foo2' is not exported
overload bar { foo1, foo2 } // OK, as 'bar' is not exported
```

If an *explicit overload* is called like a function with receiver, i.e., syntax of method call is used (see *Functions with Receiver*), then a compile-time error occurs:

```
function bar1(this: string) {}
function bar2(this: number) {}

overload bar { bar1, bar2 }

let s: string = "";
let n: number = 1;
```

(continues on next page)

(continued from previous page)

```
8
9  bar(s) // OK
10 bar(n) // OK
11 s.bar()  // Compile-time error
12 n.bar()  // Compile-time error
```

17.9.2 Explicit Class Method Overload

Explicit class method overload allows declaring a name for a set of class methods (see *Method Declarations*). The syntax is presented below:

```
explicitClassMethodOverload:
    explicitClassMethodOverloadModifier*
    'overload' identifier overloadList
    ;

explicitClassMethodOverloadModifier: 'static' | 'async';
```

The use of an *explicit class method overload* is represented in the example below:

```
class Processor {
    overload process { processNumber, processString }
    processNumber(n: number) {/*body*/}
    processString(s: string) {/*body*/}
}

let c = new C()
c.process(42) // calls processNumber
c.process("aa") // calls processString
```

The name of an *explicit method overload* can be the same as the name of a method in the same class (see *Explicit Overload Name Same As Method Name* for details).

Static explicit overload is represented in the example below:

```
class C {
    static one(n: number) {/*body*/}
    static two(s: string) {/*body*/}
    static overload foo { one, two }
}
```

A compile-time error occurs if:

- Method modifier is used more than once in an explicit method overload;
- *Identifier* in the overloaded method list refers to no accessible method (either declared or inherited) of the current class;
- *Identifier* in the overloaded method list refers to an implicitly overloaded method name;
- *Explicit overload* is:

- *Static* but the overloaded method is *non-static*;
- *Non-static* but the overloaded method is *static*;
- Marked `async` but the overloaded method is not; or
- Not `async` but the overloaded method is.

An *explicit overload* and the overloaded methods can have different access modifiers. A compile-time error occurs if an *explicit overload* is:

- `public` but at least one overloaded method is not `public`;
- `protected` but at least one overloaded method is `private`.

Valid and invalid explicit overloads are represented in the example below:

```
class C {
    private foo1(x: number) {/*body*/}
    protected foo2(x: string) {/*body*/}
    public foo3(x: boolean) {/*body*/}
    foo4() {/*body*/} // implicitly public

    public overload foo { foo3, foo4 } // OK
    protected overload bar { foo2, foo3 } // OK
    private overload goo { foo1, foo2, foo3 } // OK

    public overload err1 {foo2, foo3} // Compile-time error, foo2 is not public
    protected overload err2 {foo2, foo1} // Compile-time error, foo1 is private
}
```

Some or all overloaded functions can be `native` as follows:

```
class C {
    native foo1(x: number)
    foo2(x: string) {/*body*/}
    overload foo { foo1, foo2 }
}
```

If a superclass has an *explicit overload*, then this declaration can be overridden in a subclass. If a subclass does not override an *explicit overload*, then the overload from the superclass is inherited.

If a subclass overrides an *explicit overload*, then this declaration must list all methods of the *explicit overload* in a superclass, otherwise, a compile-time error occurs.

In addition, overriding an *explicit overload* in a subclass can include new methods and change the order of methods.

An *explicit overload* is used like an ordinary class method except that it is replaced in a call at compile time for one of overloaded methods that use the type of *object reference*. An *explicit overload* in subtypes is represented in the example below:

```
class Base {
    overload process { processNumber, processString }
    processNumber(n: number) {/*body*/}
    processString(s: string) {/*body*/}
}

class D1 extends Base {
    // method is overridden
```

(continues on next page)

(continued from previous page)

```
    override processNumber(n: number) {/*body*/}
    // overload declaration is inherited
}

class D2 extends Base {
    // method is added:
    processInt(n: int) {/*body*/}
    // new order for overloaded methods is specified:
    overload process { processInt, processNumber, processString }
}

new D1().process(1)   // calls processNumber from D1

new D2().process(1)   // calls processInt from D2 (as it is listed earlier)
new D2().process(1.0) // calls processNumber from Base (first appropriate)
```

Methods with special names (see *Indexable Types*, *Iterable Types*, and *Callable Types*) can be overloaded like ordinary methods:

```
class C {
    getByNumber(n: number): string {...}
    getByString(s: string): string {...}
    overload $_get { getByNumber, getByString }
}

let c = new C()

c[1]     // getByNumber is used
c["abc"] // getByString is used
```

If a class implements some interfaces with an *explicit overload* of the same name, then a new *explicit overload* must include all overloaded methods. Otherwise, a compile-time error occurs.

```
interface I1 {
    overload foo {f1, f2}
    // f1 and f2 are declared in I1
}
interface I2 {
    overload foo {f3, f4}
    // f3 and f4 are declared in I2
}
class C implements I1, I2 {
   // Compile-time error as no new overload is defined
}
class D implements I1, I2 {
    overload foo { f2, f3, f1, f4 } // OK, as new overload is defined
}
class E implements I1, I2 {
    overload foo { f2, f4 } // Compile-time error as not all methods are used
}

const i1: I1 = new D
```

(continues on next page)

(continued from previous page)

```
i1.foo(<arguments>) // call is valid if arguments fit first signature of {f1, f2} set

const i2: I2 = new D
i2.foo(<arguments>) // call is valid if arguments fit first signature of {f3, f4} set

const d: D = new D
d.foo(<arguments>) // call is valid if arguments fit first signature of {f2, f3, f1, f4}
↪ set
```

17.9.3 Explicit Interface Method Overload

Explicit interface method overload allows declaring a name for a set of interface methods (see *Interface Method Declarations*). The syntax is presented below:

```
explicitInterfaceMethodOverload:
    'overload' identifier overloadList
    ;
```

Each identifier in an *explicit interface method overload* list must denote exactly one accessible interface method.

A compile-time error occurs if an identifier refers to no accessible method, to a non-method entity, or to an implicitly overloaded method name.

The use of an *explicit interface method overload* is represented in the example below:

```
interface I {
    foo(): void
    bar(n?: string): void
    overload goo { foo, bar }
}

function example(i: I) {
    i.goo()        // calls i.foo()
    i.goo("hello") // calls i.bar("hello")
    i.bar()        // explicit call: i.bar(undefined)
}
```

The name of an *explicit method overload* can be the same as the name of a method in the same interface (see *Explicit Overload Name Same As Method Name* for details).

An *explicit overload* is used like an ordinary interface method, except that one of the overloaded methods replaces it in a call at compile time by using the type of *object reference*.

An *explicit interface overload* can be overridden in a class. In that case, an *explicit overload* in a class must contain all the methods overloaded in the interface. Otherwise, a compile-time error occurs.

```
class D implements I {
    foo(): void {/*body*/}
    bar(n?: string): void {/*body*/}
    overload goo( bar, foo) // order is changes
```

(continues on next page)

(continued from previous page)

```
}

let d = new D()
d.goo() // d.bar(undefined) is used, as it is the first appropriate method
```

If a class does not override an *explicit overload* declared in an interface, then it inherits the overload:

```
// Using interface overload declaration
class C implements I {
     foo(): void {/*body*/}
     bar(n?: string): void {/*body*/}
}

let c = new C()
c.goo() // calls c.foo()
```

An *explicit overload* defined in a superinterface can be overridden in a subinterface. In this case, the *overload declaration* of the subinterface must contain all methods overloaded in superinterface. Otherwise, a compile-time error occurs.

An *explicit overload* defined in a superinterface must be overridden in a subinterface if several *explicit overloads* of the same name are inherited in the interface. Otherwise, a compile-time error occurs.

```
interface I1 {
    overload foo {f1, f2}
    // f1 and f2 are declared in I1
}
interface I2 {
    overload foo {f3, f4}
    // f3 and f4 are declared in I2
}
interface I3 extends I1, I2 {
   // Compile-time error as no new overload for 'foo' is defined
}
interface I4 extends I1, I2 {
    overload foo { f4, f1, f3, f2 } // OK, as new overload is defined
}
interface I5 extends I1, I2 {
    overload foo { f1, f3 } // Compile-time error as not all methods are included
}
```

17.9.4 Explicit Overload Name Same As Method Name

The name of an *explicit overload* of a class or an interface can be the same as the name of the overloaded method. For example, a method defined in a superclass can be used as one of the overloaded methods in an *explicit overload* of the same-name subclass. This important case is represented in the following example:

```
class C {
    foo(n: number): number {/*body*/}
```

(continues on next page)

(continued from previous page)

```
}
class D extends C {
    fooString(s: string): string {/*body*/}

    overload foo {
        foo, // method 'foo' from C
        fooString
    }
}

let d = new D()
let c: C = d

d.foo(1)    // 'foo' from C is called
d.foo("aa") // 'fooString' from D is called
c.foo(1)    // method 'foo' from is called (no overload)
```

If names of a method and of an *explicit overload* are the same, then the method can be overridden as usual:

```
class C {
    foo(n: number): number {/*body*/}
}
class D extends C {
    foo(n: number): number {/*body*/} // method is overridden
    fooString(s: string): string {/*body*/}

    overload foo { foo, fooString }
}
```

This feature is also valid in interfaces, or in an interface and a class that implements the interface:

```
interface I {
    foo(n: number): number {/*body*/}
}
interface J extends I {
    fooString(s: string): string
    overload foo { foo, fooString }
}

class K implements I {
    foo(n: number): number {/*body*/}
    fooString(s: string): string {/*body*/}

    overload foo { foo, fooString }
}
```

The use of an *explicit overload* causes no ambiguity for it is considered at the call site only. An *explicit overload* name is **not** considered in the following situations:

- *Overriding*;
- List of the overloaded entities (see *Explicit Class Method Overload* and *Explicit Interface Method Overload*);
- *Method Reference*.

```
class C {
    foo(n: number): number {/*body*/}
}

class D extends C {
    fooString(s: string): string {/*body*/}

    overload foo { foo, fooString }
}

let d = new D()
let c: C = d

let func1 = c.foo // method 'foo' is used
let func2 = d.foo // method 'foo' is used
```

A compile-time error occurs if the name of an *explicit overload* is the same as the name of a method (with the same static or non-static modifier) that is not listed as an overloaded method as follows:

```
class C {
    foo(n: number) {/*body*/}
    fooString(s: string) {/*body*/}
    fooBoolean(b: boolean) {/*body*/}

    overload foo { // Compile-time error
        fooBoolean, fooString
    }
}
```

17.10 Native Functions and Methods

17.10.1 Native Functions

Native function is a function marked with the keyword `native` (see *Function Declarations*).

Native function implemented in a platform-dependent code is typically written in another programming language (e.g., *C*). A compile-time error occurs if a native function has a body.

17.10.2 Native Methods

Native method is a method marked with the keyword `native` (see *Method Declarations*).

Native methods are the methods implemented in a platform-dependent code written in another programming language (e.g., *C*).

A compile-time error occurs if:

- Method declaration contains the keyword `abstract` along with the keyword `native`.
- *Native method* has a body (see *Method Body*) that is a block instead of a simple semicolon or empty body.

17.10.3 Native Constructors

Native constructor is a constructor marked with the keyword `native` (see *Constructor Declaration*).

Native constructors are the constructors implemented in a platform-dependent code written in another programming language (e.g., *C*).

A compile-time error occurs if a *native constructor* has a non-empty body (see *Constructor Body*).

17.11 Classes Experimental

17.11.1 Final Classes

A class can be declared `final` to prevent extension, i.e., a class declared `final` can have no subclasses. No method of a `final` class can be overridden.

If a class type `F` expression is declared *final*, then only a class `F` object can be its value.

A compile-time error occurs if the `extends` clause of a class declaration contains another class that is `final`.

17.11.2 Final Methods

A method can be declared `final` to prevent it from being overridden (see *Overriding Methods*) in subclasses.

A compile-time error occurs if:

- The method declaration contains the keyword `abstract` or `static` along with the keyword `final`.
- A method declared `final` is overridden.

17.12 Default Interface Method Declarations

The syntax of *interface default method* is presented below:

```
interfaceDefaultMethodDeclaration:
    'private'? identifier signature block
    ;
```

A default method can be explicitly declared `private` in an interface body.

A block of code that represents the body of a default method in an interface provides a default implementation for any class if such a class does not override the method that implements the interface.

17.13 Adding Functionality to Existing Types

ArkTS supports adding functions and accessors to already defined types. The usage of functions so added looks the same as if they are methods and accessors of such types. The mechanism is called *Functions with Receiver*. This feature is often used to add new functionality to a class or an interface without having to inherit from the class or to implement the interface. However, it can be used not only for classes and interfaces but also for other types.

Moreover, *Function Types with Receiver* and *Lambda Expressions with Receiver* can be defined and used to make the code more flexible.

17.13.1 Functions with Receiver

Function with receiver declaration is a top-level declaration (see *Top-Level Declarations*) that looks almost the same as *Function Declarations*, except that the first mandatory parameter uses keyword `this` as its name.

The syntax of *function with receiver* is presented below:

```
functionWithReceiverDeclaration:
    'function' identifier typeParameters? signatureWithReceiver block
    ;

signatureWithReceiver:
    '(' receiverParameter (', ' parameterList)? ')' returnType?
    ;

receiverParameter:
    annotationUsage? 'this' ':' type
    ;
```

Function with receiver can be called in the following two ways by making:

- Ordinary function call (see *Function Call Expression*) when the first argument is the receiver object;
- Method call (see *Method Call Expression*) when the receiver is an `objectReference` before the function name passed as the first argument of the call.

All other arguments are handled in an ordinary manner.

The keyword `this` must be used in the parameter list for the first parameter only. If it is used for other parameters, then a compile-time error occurs.

The keyword `this` can be used inside a *function with receiver* where it corresponds to the first parameter. The type of parameter `this` is called *receiver type* (see *Receiver Type*):

```
class A {
  num: number = 1
  foo(): void { console.log(this.num); }
}
function bar(this: A) {
  this.num = 5
}
let a = new A()
a.foo() // method is called
a.bar() // Function with receiver is called
a.foo() // method is called
```

The first parameter named `this` is readonly.

If the *receiver type* is a class or interface type, then `private` or `protected` members are not accessible (see *Accessible*) within the body of a *function with receiver*. Only `public` members can be accessed:

```
class A {
    foo () { ... this.bar() ... }
                // function bar() is accessible here
    protected member_1 ...
    private member_2 ...
}
function bar(this: A) { ...
   this.foo() // Method foo() is accessible as it is public
   this.member_1 // Compile-time error as member_1 is not accessible
   this.member_2 // Compile-time error as member_2 is not accessible
   ...
}
let a = new A()
a.foo() // Ordinary class method is called
a.bar() // Function with receiver is called
```

Derived classes or interfaces can be used as receivers:

```
class C {}

function foo(this: C) {}
function bar(this: C, n: number): void {}

let c = new C()

// as a function call:
foo(c)
bar(c, 1)

// as a method call:
c.foo()
c.bar(1)

interface D {}
function foo1(this: D) {}
function bar1(this: D, n: number): void {}

```

(continues on next page)

(continued from previous page)

```
function demo (d: D) {
   // as a function call:
   foo1(d)
   bar1(d, 1)

   // as a method call:
   d.foo1()
   d.bar1(1)
}

class E implements D {}
const e = new E

// derived class is used as a receiver for a method call:
e.foo1()
e.bar1(1)

// the same as a function call:
foo1(e)
bar1(e, 1)
```

Function with receiver can be generic as in the following example:

```
class G<T> {}

function foo<T>(this: G<T>, p: T) {
    console.log (p)
}

class C {}

let g = new G<C>
g.foo(new C)    // implicit instantiation
g.foo<C>(new C) // explicit instantiation
```

When the receiver type contains an accessible instance method (see *Accessible*) with the same name as the function with receiver, the instance method has a priority over the implicitly called function with receiver. The function with receiver still can be called explicitly:

```
class A {
    foo (): int { return 1; }
}

function foo(this: A): int { return 2; }

console.log((new A).foo())  // instance method called, prints '1'
console.log(foo(new A)) // explicit call of a receiver function, prints '2'
```

Functions with receiver are dispatched statically. What function is being called is known at compile time based on the receiver type specified in the declaration. A *function with receiver* can be applied to the receiver of any derived class until it is overridden within the derived class:

```
class Base { ... }
class Derived extends Base { ... }

function foo(this: Base) { console.log ("Base.foo is called") }

let b: Base = new Base()
b.foo() // `Base.foo is called` to be printed
b = new Derived()
b.foo() // `Base.foo is called` to be printed
```

A *function with receiver* can be defined in a module other than the one that defines the receiver type. This is represented in the following example:

```
// file a.ets
class A {
    foo() { ... }
}

// file ext.ets
import {A} from "a.ets" // name 'A' is imported
function bar(this: A) () {
   this.foo() // Method foo() is called
}
```

A *function with receiver* can be defined in a namespace (see *Namespace Declarations*). A *function with receiver* cannot be called by using the *method call* syntax outside the namespace, though, because an entity exported from a namespace can be accessed in the form of a `qualifiedName` only.

This situation is represented in the following example:

```
namespace NS {
    export function foo(this: int) {}
    function bar(i: int) {
        i.foo() // OK, method call is used
    }
}

let i = 1
NS.foo(i)  // OK, function call is used
i.foo()    // Compile-time error, 'foo' is not resolved
i.NS.foo() // Compile-time error, 'NS' is not defined for 'int'
```

Note

While a function with receiver can be used in an explicit overload list, such an overload cannot be called by using the method access syntax as in the example provided in *Explicit Function Overload*.

17.13.2 Receiver Type

Receiver type is the type of the *receiver parameter* in a function, function type, and lambda with receiver. A *receiver type* can be an interface type, a class type, or an array type. Otherwise, a compile-time error occurs.

Using array type as a *receiver type* is presented in the example below:

```
function addElements(this: number[], ...s: number[]) {
 ...
}

let x: number[] = [1, 2]
x.addElements(3, 4)
```

17.13.3 Function Types with Receiver

Function type with receiver specifies the signature of a function or lambda with receiver. It is almost the same as *function type* (see *Function Types*), except that the first parameter is mandatory, and the keyword `this` is used as its name:

The syntax of *function type with receiver* is presented below:

```
functionTypeWithReceiver:
    '(' receiverParameter (',' ftParameterList)? ')' ftReturnType
    ;
```

The type of a *receiver parameter* is called the *receiver type* (see *Receiver Type*).

```
class A {...}

type FA = (this: A) => boolean
type FN = (this: number[], max: number) => number
```

Function type with receiver can be generic as in the following example:

```
class B<T> {...}

type FB<T> = (this: B<T>, x: T): void
type FBS = (this: B<string>, x: string): void
```

The usual rule of function type compatibility (see *Subtyping for Function Types*) is applied to *function type with receiver*, and parameter names are ignored.

```
class A {...}

type F1 = (this: A) => boolean
type F2 = (a: A) => boolean

function foo(this: A): boolean {}
function goo(a: A): boolean {}

```

(continues on next page)

(continued from previous page)

```
  let f1: F1 = foo // OK
  f1 = goo // OK

  let f2: F2 = goo // OK
  f2 = foo // OK
  f1 = f2 // OK
```

The sole difference is that only an entity of *function type with receiver* nut not an entity of a compatible *function type* can be used in *Method Call Expression*.

```
  let a = new A()
  a.f1() // OK, function type with receiver
  f1(a)  // OK

  a.f2() // Compile-time error
  f2(a) // OK
```

Note

The limitation of the method call syntax can be easily bypassed by assigning an ordinary function to a compatible *function type with receiver*. A snippet of code illustrative of parameter type with receiver is represented by the example below.

Function type with receiver can be used as a parameter type. Using parameter type with receiver is represented by the example below:

```
function foo(p: number, f: (this: number)=> number) {
    console.log(p.f(), f(p))
}

function goo(this: number) { return this - 1 }
function bar(this: number) { return this + 1 }
function compat(n: number) { return n }

let n: number = 1
foo(n, goo)  // prints `0 0`
foo(n, bar)  // prints `2 2`
foo(n, compat)  // prints `1 1`
```

The method call syntax cannot be used when assigning the actual entity to a variable of *function type with receiver*. Attempting to do so causes a compile-time error:

```
function foo<T extends Object>(this: T, functor: (this: T)=> void): void {
   // following two calls are equivalent
   functor(this)
   this.functor()
}

function bar<T>(this: T): void {
   console.log(this)
}
```

(continues on next page)

(continued from previous page)

```
let x = 5
x.foo(bar<int>) // OK
let y = bar<int> // OK
x.foo(y) // OK

// compile time error - can not assign entity with method call syntax
// to a function type
x.foo(x.bar)
x.foo(x.bar<int>)
let z = x.bar
let y = x.bar<int>
```

17.13.4 Lambda Expressions with Receiver

Lambda expression with receiver defines an instance of a *function type with receiver* (see *Function Types with Receiver*). It looks almost the same as an ordinary lambda expression (see *Lambda Expressions*), except that the first parameter is mandatory, and the keyword `this` is used as its name:

The syntax of *lambda expression with receiver* is presented below:

```
lambdaExpressionWithReceiver:
    annotationUsage?
    '(' receiverParameter (',' lambdaParameterList)? ')'
    returnType? '=>' lambdaBody
    ;
```

The use of annotations is discussed in *Using Annotations*.

The keyword `this` can be used inside a *lambda expression with receiver*, It corresponds to the first parameter:

```
class A { name = "Bob" }

let show = (this: A): void {
    console.log(this.name)
}
```

Lambda can be called in two syntactical ways represented by the example below:

```
class A {
  name: string
  constructor (n: string) {
      this.name = n
  }
}

function foo(aa: A[], f: (this: A) => void) {
  for (let a of aa) {
      a.f() // first way
```

(continues on next page)

(continued from previous page)

```
        f (a) // second way
    }
  }

  let aa: A[] = [new A("aa"), new A("bb")]
  foo(aa, (this: A) => { console.log(this.name)} ) // output: "aa" "bb"
```

Note

If *lambda expression with receiver* is declared in a class or interface, then `this` use in the lambda body refers to the first lambda parameter and not to the surrounding class or interface. Any lambda call outside a class has to use the ordinary syntax of arguments as represented by the example below:

```
  class B {
    foo() { console.log ("foo() from B is called") }
  }
  class A {
    foo() { console.log ("foo() from A is called") }
    bar() {
        let lambda1 = (this: B): void => { this.foo() } // local lambda
        new B().lambda1()
    }
    lambda2 = (this: B): void => { this.foo() } // class field lambda
  }
  new A().bar() // Output is 'foo() from B is called'
  new A().lambda2 (new B) // Argument is to be provided in its usual place

  interface I {
     lambda: (this: B) => void // Property of the function type
  }
  function foo (i: I) {
     i.lambda(new B) // Argument is to be provided in its usual place
  }
```

17.14 Trailing Lambdas

The *trailing lambda* is a special form of notation for function or method call when the last parameter of a function or a method is of function type, and the argument is passed as a lambda using the *Block* notation. The *trailing lambda* syntactically looks as follows:

```
trailingLambda:
    'async'? block
    ;
```

The modifier `async` is used optionally to mark *Asynchronous Lambdas*.

The use of a trailing lambda is represented in the example below:

```
class A {
    foo (f: ()=>void) { ... }
}

let a = new A()
a.foo() { console.log ("method lambda argument is activated") }
// method foo receives last argument as the trailing lambda
```

Currently, no parameter can be specified for the type of a trailing lambda, except a receiver parameter (see *Lambda Expressions with Receiver*). Otherwise, a compile-time error occurs.

A block immediately after a call is always handled as *trailing lambda*. A compile-time error occurs if the last parameter of the called entity is not of a function type.

The semicolon ';' separator can be used between a call and a block to indicate that the block does not define a *trailing lambda*. When calling an entity with the last optional parameter (see *Optional Parameters*), it means that the call must use the default value of the parameter.

```
function foo (f: ()=>void) { ... }

foo() { console.log ("trailing lambda") }
// 'foo' receives last argument as the trailing lambda

function bar(f?: ()=>void) { ... }

bar() { console.log ("trailing lambda") }
// function 'bar' receives last argument as the trailing lambda,
bar(); { console.log ("that is the block code") }
// function 'bar' is called with parameter 'f' set to 'undefined'

function goo(n: number) { ... }

goo() { console.log("aa") } // Compile-time error as goo() requires an argument
goo(); { console.log("aa") } // Compile-time error as goo() requires an argument
```

If there are optional parameters in front of an optional function type parameter, then calling such a function or method can skip optional arguments and keep the trailing lambda only. This implies that the value of all skipped arguments is `undefined`.

```
function foo (p1?: number, p2?: string, f?: ()=>string) {
    console.log (p1, p2, f?.())
}

foo()                           // undefined undefined undefined
foo() { return "lambda" }       // undefined undefined lambda
foo(1) { return "lambda" }      // 1 undefined lambda
foo(1, "a") { return "lambda" } // 1 a lambda
```

17.15 Accessor Declarations

Accessor is either a top-level declaration (see *Top-Level Declarations*) or a declaration inside a namespace (see *Namespace Declarations*) that declares a getter, a setter, or functions with predefined signatures. The syntatic form of accessor usage mimics code patterns used to work with variables, i.e., getting or setting a variable value.

The syntax of *accessor declarations* is presented below:

```
accessorDeclaration:
    'native'?
    ( 'get' identifier '(' ')' returnType? block?
    | 'set' identifier '(' requiredParameter ')' block?
    )
    ;
```

The modifier `native` indicates that the accessor is a *native accessor* (similarly to *Native Functions*).

A non-native accessor must have a body. A compile-time error occurs if:

- Native accessor has a body; or
- Non-native accessor has no body.

A *get-accessor* (*getter*) must have an explicit return type and no parameters, or no return type at all on condition the type can be inferred from the getter body (see *Return Type Inference*). A *set-accessor* (*setter*) must have a single parameter and no return type.

> **Note**
>
> If an *accessor* is an entity of a namespace, then the same rules apply to it when exporting and using qualified names as the rules that apply to other namespace entities (see *Namespace Declarations*).

A compile-time error occurs if:

- Getter or setter is used in a call expression (like a function);
- Getter return type cannot be inferred from the getter body; or
- *Set-accessor* (*setter*) has an optional parameter (see *Optional Parameters*):

The typical use of an accessor to control value setting is represented in the following example:

```
let saved_age = 0

export get age(): number { return saved_age }
export set age(a: number) {
    if (a < 0) { throw new Error("wrong age") }
    saved_age = a
}
```

Which accessor (getter or setter) is to be called is defined by the place of use:

```
get name(): string { return "" }
set name(x: string) { }

console.log (name) // Getter is called
name = "some string" // Setter is called
```

However, an accessor declaration must be distinguishable from other entities, and a compile-time error occurs if:

- Accessor name is the same as that of another entity in a scope;
- Names of two getters or two setters in a scope are the same.

```
let name = "Bob"
get name(): string { return "Alice" } // Compile-time error
```

No additional restrictions are imposed on signatures of getters and setters that have the same name.

```
set hashCode(x: string) {/*body*/}
get hashCode(): long {/*body*/} // OK

hashCode = "some string"
const l: long = hashCode
```

The use of getters and setters looks identical to the use of variables. A compile-time error occurs if:

- Getter is used in the position of a *left-hand-side expression* in an *Assignment*;
- Setter is used to get a value.

```
get magicNumber(): number { return 42 }
set randomSeed(a: number) {}

console.log(magicNumber) // OK, getter is used
magicNumber = 15 // Compile-time error, setter is not defined

randomSeed = 42 // OK, setter is used
console.log(randomSeed) // Compile-time error, getter is not defined
```

Accessors can be declared at all places where *Top-Level Declarations* including namespaces can be used:

```
namespace N {
    let saved_age = 0

    export get age(): number { return saved_age }
    export set age(a: number) {
        if (a < 0) { throw new Error("wrong age") }
        saved_age = a
    }
}

N.age = 18
console.log(N.age)
```

17.16 Pattern Matching

Pattern Matching is a set of powerful features supported by most modern programming languages. *Pattern matching* generally allows checking a value against a pattern, and executing a corresponding action after a match is successful. A successful match can also deconstruct a value into its constituent parts.

The current version of ArkTS supports a simple *pattern matching* feature called *destructuring assignment*. Other features are to be added in the forthcoming revisions of this specification.

17.16.1 Destructuring Assignment

Destructuring assignment allows extracting values from arrays or tuples, and assigning them to distinct variables.

The syntax of a *destructuring assignment* is as follows:

```
destructuringAssignment:
    '[' lhsExpression? (',' lhsExpression?)* ']' '=' rhsExpression
    ;
```

The definitions of `lhsExpression` and `rhsExpression` are provided in *Assignment*.

`rhsExpression` must be of array type or tuple type. Otherwise, a compile-time error occurs.

Destructuring assignment can be considered a compact form of a set of assignments (see *Simple Assignment Operator*). Items in a *left-hand-side* expression (whether `lhsExpression` or none) correspond to the sequence of elements in `rhsExpression` staring from the first element (i.e., the element with the index `0`) of an array or a tuple:

```
function foo(x: string[]) {
    let a = ""
    let b = "";
    [a, , b] = x
    // this line works the same as the previous line:
    a = x[0]; b = x[2]
}
```

In the example above, `a` takes the value `x[0]`, and `b` takes the value `x[2]`. If an attempt is made to have an array element with an index greater than or equal to the length of the array, then *RangeError* is thrown in exactly the same way as in *Array Indexing Expression*.

If `lhsExpression` is missing, then the corresponding element of an array or of a tuple is ignored.

A compile-time error occurs if:

- Type of an array element or of a tuple element is not assignable to the type of a corresponding `lhsExpression` (see *Assignability*);
- `rhsExpression` is of a tuple type, and `lhsExpression` corresponds to the missing tuple element.

Valid and erroneous destructuring is represented in the following example:

```
function foo(x: [string, number, string]) {
    let a: string
    let b: number
    [a] = x; // OK
    [, b, a] = x; // OK
    [, b, a,,,,] = x; // OK
    [b] = x; // Compile-type error, x[0] is not assignable to 'b'
    [, b] = x; // OK
    [,,,b] = x; // Compile-time error, there is no element for variable 'b'
}
```

CHAPTER

EIGHTEEN

ANNOTATIONS

Annotation is a special language element that changes the semantics of the declaration to which it is applied by adding metadata.

Declaring and using an annotation is represented in the example below:

```
// Annotation declaration:
@interface ClassAuthor {
    authorName: string
}

// Annotation use:
@ClassAuthor({authorName: "Bob"})
class MyClass {/*body*/}
```

The annotation *ClassAuthor* in the example above adds metadata to the class declaration.

An annotation must be placed immediately before the declaration to which it is applied. An annotation can include arguments as in the example above.

For an annotation to be used, the name of the annotation must be prefixed with the character '`@`' (e.g., `@MyAnno`). No white space and line separator is allowed between the character '`@`' and the name:

```
ClassAuthor({authorName: "Bob"}) // Compile-time error, no '@'
@ ClassAuthor({authorName: "Bob"}) // Compile-time error, space is forbidden
```

A compile-time error occurs if the annotation name is not accessible (see *Accessible*) at the place of use. An annotation declaration can be exported and used in other modules.

Multiple annotations can be applied to a single declaration:

```
@MyAnno()
@ClassAuthor({authorName: "John Smith"})
class MyClass {/*body*/}
```

18.1 Declaring Annotations

Declaring an *annotation* is similar to declaring an interface where the keyword `interface` is prefixed with the character `'@'`.

The syntax of *annotation declaration* is presented below:

```
annotationDeclaration:
    '@interface' identifier '{' annotationField* '}'
    ;
annotationField:
    identifier ':' type constInitializer?
    ;
constInitializer:
    '=' constantExpression
    ;
```

As any other declared entity, an annotation can be exported by using the keyword `export`.

Type in the annotation field is restricted (see *Types of Annotation Fields*).

The default value of an *annotation field* can be specified by using *initializer* as *constant expression*. A compile-time error occurs if the value of this expression cannot be evaluated at compile time.

Annotation must be defined at the top level. Otherwise, a compile-time error occurs.

Annotation cannot be extended as inheritance is not supported.

The name of an *annotation* cannot coincide with the name of another entity:

```
@interface Position {/*properties*/}

class Position {/*body*/} // Compile-time error, duplicate identifier
```

An annotation declaration defines no type. No type alias can be applied to the annotation or used as an interface:

```
@interface Position {}
type Pos = Position // Compile-time error

class A implements Position {} // Compile-time error
```

18.1.1 Types of Annotation Fields

The choice of *types for annotation fields* is limited to the following:

- *Numeric Types*;
- Type `boolean` (see *Type boolean*);
- *Type string*;
- Enumeration types (see *Enumerations*);
- Array of the above types (e.g., `string[]`), including arrays of arrays (e.g., `string[][]`).

A compile-time error occurs if any other type is used as the type of an *annotation field*.

18.2 Using Annotations

The following syntax is used to apply an annotation to a declaration, and to define the values of annotation fields:

```
annotationUsage:
    AnnotationUsageNoParentheses |
    annotationUsageWithParentheses
    ;

annotationUsageNoParentheses:
    '@' qualifiedName
    ;

annotationUsageWithParentheses:
    '@' qualifiedName annotationValues
    ;
annotationValues:
    '(' (objectLiteral | constantExpression)? ')'
    ;
```

An annotation declaration is represented in the example below:

```
@interface ClassPreamble {
    authorName: string
    revision: number = 1
}
@interface MyAnno{}
```

In general, annotation field values are set by an *object literal*. In a special case, an annotation field value is set by using an expression (see *Using Single Field Annotations*).

A value for an annotation field must be:

- a constant expression, if the field is not of an array type; or
- an *Array Literal* with elements that are either constant expressions or, in case of array of array, enclosed array literals with constant expressions.

Otherwise, a compile-time error occurs.

The use of annotation is presented in the example below. The annotations in this example are applied to class declarations:

```
@ClassPreamble({authorName: "John", revision: 2})
class C1 {/*body*/}

@ClassPreamble({authorName: "Bob"}) // default value for revision = 1
class C2 {/*body*/}

@MyAnno()
class C3 {/*body*/}
```

Annotations can be applied to the following:

- *Top-Level Declarations*;
- Class members (see *Class Members*) except overridden fields (see the example below);

- Interface members (see *Interface Members*);
- Type usage (see *Using Types*);
- Parameters (see *Parameter List* and *Optional Parameters*);
- Lambda expression (see *Lambda Expressions* and *Lambda Expressions with Receiver*);
- *Constant Or Variable Declarations*.

Otherwise, a compile-time error occurs:

```
function foo () @MyAnno() {} // wrong target for annotation
```

A compile-time error occurs if an annotation is applied to an overridden field (see *Override Fields*):

```
class C {
    field: int = 1
}
class D extends C {
    @MyAnno() // Compile-time error
    field: int = 2
}
```

Repeatable annotations are not supported, i.e., an annotation can be applied to an entity no more than once:

```
@ClassPreamble({authorName: "John"})
@ClassPreamble({authorName: "Bob"}) // Compile-time error
class C {/*body*/}
```

When using an annotation, the order of values has no significance:

```
@ClassPreamble({authorName: "John", revision: 2})
// the same as:
@ClassPreamble({revision: 2, authorName: "John"})
```

When using an annotation, all fields without default values must be listed. Otherwise, a compile-time error occurs:

```
@ClassPreamble() // Compile-time error, authorName is not defined
class C1 {/*body*/}
```

If a field of an array type for an annotation is defined, then its value is set by using the array literal syntax:

```
@interface ClassPreamble {
    authorName: string
    revision: number = 1
    reviewers: string[]
}

@ClassPreamble(
    {authorName: "Alice",
    reviewers: ["Bob", "Clara"]}
)
class C3 {/*body*/}
```

If setting annotation properties is not required, then parentheses can be omitted after the annotation name:

```
@MyAnno
class C4 {/*body*/}
```

18.2.1 Using Single Field Annotations

If annotation declaration defines only one field, then it can be used with a short notation to specify just one expression instead of an object literal:

```
@interface deprecated{
    fromVersion: string
}

@deprecated("5.18")
function foo() {}

@deprecated({fromVersion: "5.18"})
function goo() {}
```

A short notation and a notation with an object literal behave in exactly the same manner.

18.3 Exporting and Importing Annotations

An annotation can be exported and imported. However, a few forms of export and import directives are supported.

An annotation declaration to be exported must be marked with the keyword `export` as follows:

```
// a.ets
export @interface MyAnno {}
```

If an annotation is imported as a part of an imported module, then the annotation is accessed by its qualified name:

```
// b.ets
import * as ns from "./a"

@ns.MyAnno
class C {/*body*/}
```

Unqualified import is also allowed:

```
// b.ets
import { MyAnno } from "./a"

@MyAnno
class C {/*body*/}
```

An annotation declaration does not define a type. Using `export type` or `import type` notations to export or import an annotation causes a compile-time error:

```
import type { MyAnno } from "./a" // Compile-time error
```

If annotations are used in the following cases, then a compile-time error also occurs:

- Export default,
- Import default,
- Rename in export, and
- Rename in import.

```
import {MyAnno as Anno} from "./a" // Compile-time error
```

18.4 Ambient Annotations

The syntax of *ambient annotations* is presented below:

```
ambientAnnotationDeclaration:
    'declare' annotationDeclaration
    ;
```

Such a declaration does not introduce a new annotation but provides type information that is required to use the annotation. The annotation itself must be defined elsewhere. A runtime error occurs if no annotation corresponds to the ambient annotation used in the program.

An ambient annotation and the annotation that implements it must be exactly identical, including field initialization:

```
// a.d.ets
export declare @interface NameAnno{name: string = ""}

// a.ets
export @interface NameAnno{name: string = ""} // OK
```

The code in the example below is incorrect because the ambient declaration is not identical to the annotation declaration:

```
// a.d.ets
export declare @interface VersionAnno{version: number} // initialization is missing

// a.ets
export @interface VersionAnno{version: number = 1}
```

An ambient declaration can be imported and used in exactly the same manner as a regular annotation:

```
// a.d.ets
export declare @interface MyAnno {}

// b.ets
import { MyAnno } from "./a"
```

(continues on next page)

(continued from previous page)

```
@MyAnno
class C {/*body*/}
```

If an annotation is applied to an ambient declaration in the *.d.ets* file (see the example below), then the annotation is to be applied to the implementation declaration manually, because the annotation is not automatically applied to the declaration that implements the ambient declaration:

```
// a.d.ets
export declare @interface MyAnno {}

@MyAnno
declare class C {}
```

18.5 Standard Annotations

Standard annotation is an annotation that is defined in *Standard Library*, or implicitly defined in the compiler (*built-in annotation*). *Standard annotation* is usually known to the compiler. It modifies the semantics of the declaration it is applied to.

An annotation that annotates a declaration of another annotation is called *meta-annotation*.

18.5.1 Retention Annotation

`@Retention` is a standard *meta-annotation* that is used to annotate a declaration of another annotation. A compile-time error occurs if it is used in other places.

The annotation has a single field `policy` of type `string`. It is typically used as follows:

```
@Retention({policy: "RUNTIME"})
@interface MyAnno {} // this annotation uses "RUNTIME" policy

@MyAnno //
class C {}
```

The value of this field determines at which point an annotation is used, and discarded after use. The retention policies can be of three types:

- "SOURCE"

  Annotations that use "SOURCE" policy are processed at compile time, and are discarded after compilation;

- "BYTECODE"

  Metadata specified in annotations that use "BYTECODE" policy are saved into the bytecode file, but are discarded at runtime.

- "RUNTIME"

  Metadata specified in annotations that use "RUNTIME" policy are saved into the bytecode file, are retained and can be accessed at runtime.

The default retention policy is "BYTECODE".

A compile-time error occurs if any other string literal is used as the value of `policy` field.

As `@Retention` has a single field, it can be used with a short notation (see *Using Single Field Annotations*) as follows:

```
@Retention("SOURCE")
@interface Author {name: string} // this annotation uses "SOURCE" policy
```

18.5.2 Target Annotation

`@Target` is a standard *meta-annotation* that is used to annotate a declaration of another annotation. A compile-time error occurs if `@Target` is used elsewhere.

`@Target` specifies the set of source code contexts in which the declared annotation can be used. The contexts are specified by using a set of values of an `AnnotationTargets` enumeration defined in *Standard Library*.

The annotation `@Target` has a single field `targets` of type `AnnotationTargets[]`. It is typically used as follows:

```
// short form:
@Target(["FUNCTION", "CLASS_METHOD"])
@interface SpecialCall {/*some fields*/}

// long form:
@Target({targets: ["PARAMETER"]})
@interface SpecialParameter {/*some fields*/}
```

If the annotation is present in the declaration of annotation `X`, then the compiler checks that `X` is used in the specified contexts only. Otherwise, a compile-time error occurs.

If no annotation is present in the declaration of annotation `X`, then the usage of `X` is not restricted.

An `AnnotationTargets` union type contains string literals for the following targets:

- Targets for *Top-Level Declarations*:
  - `"CLASS"`;
  - `"ENUMERATION"`;
  - `"FUNCTION"`;
  - `"FUNCTION_WITH_RECEIVER"`;
  - `"INTERFACE"`;
  - `"NAMESPACE"`;
  - `"TYPE_ALIAS"`;
  - `"VARIABLE"`;
- Targets for *Class Members*:
  - `"CLASS_FIELD"`;

  - "CLASS_METHOD";
  - "CLASS_GETTER";
  - "CLASS_SETTER";
- Targets for *Interface Members*:
  - "INTERFACE_METHOD";
  - "INTERFACE_GETTER";
  - "INTERFACE_SETTER";
- Other targets:
  - "LAMBDA" for *Lambda Expressions* and *Lambda Expressions with Receiver*;
  - "PARAMETER" for function, method, and lambda parameter;
  - "STRUCT" (see *Keyword struct and ArkUI*);
  - "TYPE" (see *Using Types*).

A compile-time error occurs if an enumeration member is used more than once in an `@Target` annotation:

```
@Target(["CLASS", "INTERFACE", "CLASS"]) // Compile-time error
@interface Anno {}
```

18.6 Runtime Access to Annotations

For an annotation with *retention policy* (see *Retention Annotation*) `BYTECODE` or `RUNTIME` an abstract class with the name of the annotation is implicitly declared. All fields of this class are `readonly`. If a field is of an array type, the array type is also `readonly`.

For the following annotation:

```
@Retention("RUNTIME")
@interface MyAnno {
    name: string
    attrs: number[]
}
```

–the abstract class is declared:

```
abstract class MyAnno {
    readonly name: string
    readonly attrs: readonly number[]
}
```

The use of such a class is represented in following example:

```
@MyAnno({name: "someName", attr: [1, 2]})
class A {}

```

(continues on next page)

(continued from previous page)

```
let my: MyAnno = // call of reflection library to get instance of annotation for type A
console.log(my.name) // output: someName
```

CHAPTER

NINETEEN

STANDARD LIBRARY

The Standard Library of the ArkTS language defines the required set of types, variables, constants, functions, and annotations that provide APIs for effective and convenient programming.

A detailed description of all elements of the standard library is covered in a separate document that is a part of the ArkTS distribution package. This document also defines a set of exported entities directly accessible in all ArkTS modules.

CHAPTER

TWENTY

IMPLEMENTATION DETAILS

Important implementation details are discussed in this section.

20.1 Import Path Lookup

If an import path `<some path>/name` is resolved to a path in the folder '*name*', then the compiler executes the following lookup sequence:

- If the folder contains the file `index.ets`, then this file is imported as a module written in ArkTS;
- If the folder contains the file `index.ts`, then this file is imported as a module written in TypeScript.

20.2 Modules in Host System

Modules are created and stored in a manner that is determined by the host system. The exact manner modules are stored in a file system is determined by a particular implementation of the compiler and other tools.

A simple implementation stores every module in a single file.

20.3 Getting Runtime Type Via Reflection

The ArkTS standard library (see *Standard Library*) provides a pseudogeneric static method `Class.from<T>()` to be processed by the compiler in a specific way during compilation. A call to this method allows getting a value of type `Class` that represents type `T` at runtime.

```
let type_of_int: Class = Class.from<int>()
let type_of_string: Class = Class.from<string>()
let type_of_number: Class = Class.from<number>()
let type_of_Object: Class = Class.from<Object>()

class UserClass {}
let type_of_user_class: Class = Class.from<UserClass>()

```

(continues on next page)

(continued from previous page)

```
interface SomeInterface {}
let type_of_interface: Class = Class.from<SomeInterface>()
```

If type `T` used as type argument is affected by *Type Erasure*, then the function returns value of type `Class` for *effective type* of `T` but not for `T` itself:

```
let type_of_array1: Class = Class.from<int[]>() // value of Class for Array<>
let type_of_array2: Class = Class.from<Array<number>>() // the same Class value
```

20.4 Ensuring Module Initialization

The ArkTS standard library (see *Standard Library*) provides a top-level function `initModule()` with a single parameter of type `string`. Its value is resolved at compile time according to the rules of import path resolution (see *ImportPath Resolution Rules*). A call to this function ensures that the module referred by the argument is available, and that its initialization (see *Static Initialization*) is performed. An argument must be a string literal. Otherwise, a compile-time error occurs.

The current module has no access to the exported declarations of the module referred by the argument. If such module is not available or any other runtime issue occurs then a proper exception is thrown.

```
initModule ("@ohos/library/src/main/ets/pages/Index")
```

20.5 Generic and Function Types Peculiarities

The current compiler and runtime implementations use type erasure. Type erasure affects the behavior of generics and function types. It is expected to change in the future. A particular example is provided in the last bullet point in the list of compile-time errors in *instanceof Expression*.

20.6 Keyword `struct` and ArkUI

The current compiler reserves the keyword `struct` because it is used in legacy ArkUI code. This keyword can be used as a replacement for the keyword `class` in *Class Declarations*. Class declarations marked with the keyword `struct` are processed by the ArkUI plugin and replaced with class declarations that use specific ArkUI types.

20.7 Make a Bridge Method for Overriding Method

Situations are possible where the compiler must create an additional bridge method to provide a type-safe call for the overriding method in a subclass of a generic class. Overriding is based on *erased types* (see *Type Erasure*). The situation is represented in the following example:

```
class B<T extends Object> {
    foo(p: T) {}
}
class D extends B<string> {
    foo(p: string) {} // original overriding method
}
```

In the example above, the compiler generates a *bridge* method with the name `foo` and signature `(p: Object)`. The *bridge* method acts as follows:

- Behaves as an ordinary method in most cases, but is not accessible from the source code, and does not participate in overloading;
- Applies narrowing to argument types inside its body to match the parameter types of the original method, and invokes the original method.

The use of the *bridge* method is represented by the following code:

```
let d = new D()
d.foo("aa") // original method from 'D' is called
let b: B<string> = d
b.foo("aa") // bridge method with signature (p: Object) is called
// its body calls original method, using (p as string) to check the type of the argument
```

More formally, a bridge method $m(C_1, \ldots, C_n)$ is created in `D`, in the following cases:

- Class `B` comprises type parameters $B<T_1$ `extends` $C_1, \ldots, T_n$ `extends` $C_n>$;
- Subclass `D` is defined as `class D extends` $B<X_1, \ldots, X_n>$;
- Method `m` of class `D` overrides `m` from `B` with type parameters in signature, e.g., $(T_1, \ldots, T_n)$;
- Signature of the overridden method `m` is not $(C_1, \ldots, C_n)$.

CHAPTER

TWENTYONE

RUNTIME SYSTEM

21.1 Overview

Runtime specification presents complete information on the structure of the ArkTS runtime system with the detailed description of its binary file structure, type system and public interfaces.

21.2 Fully Qualified Name

Fully qualified name is used to create unique identifiers for different types and entities at runtime and in binary files. A *Fully qualified name* of an entity is generated from its *unqualified name* by a build system, and specific rules vary from one build system to another. However, the ArkTS runtime expects concrete *fully qualified name* for some predefined types. Such cases are described in *Language Representation In Binary File*.

21.3 Binary File Format

The ArkTS binary file format is designed with the following goals in mind:

- Compactness,
- Fast information access,
- Low memory footprint,
- Extensibility, and
- Compatibility.

Compactness. Many mobile applications use numerous types, methods, and fields. The number is very large, and does not fit into a 16-bit unsigned integer. As a result, an application developer creates several files, and some data cannot be deduplicated.

Current binary file format must extend these limits to conform to modern requirements. It is achieved by making all direct references in a binary file 4 bytes long to allow 4Gb addressing.

16-bit indices are used to refer to classes, methods and fields in bytecode and metadata for greater compactness. A file can contain multiple indices, each covering a part of a file as described in *RegionHeader*.

A file format uses *TaggedValue* to enable compact value storage to store only the information available and avoid zero offset to any missing information.

Fast information access. The binary file format must support fast access to information. It means that redundant references are to be avoided. The binary file format must also avoid data indices (e.g., sorted list of strings) where possible. However, the current binary file format supports one index that is a sorted list of offsets to classes. This compact index allows finding type definitions quickly as the binary file is loaded.

The current binary format distinguishes between two types of classes, fields, and methods:

- *Foreign* entities are referenced in the current binary file but defined in other binary files.
- *Local* entities are defined within the current binary file.

As a local* entity and a corresponding *foreign* entity have the same header, offsets to *foreign* and *local* entities can be used interchangeably when a class, a field, or a method is referenced in a binary file.

A class, field or method type can be deduced at runtime by checking whether its offset belongs to a foreign region (`[foreign_off; foreign_off + foreign_size)`) or not. The definitions of *foreign* entities are found in other binary files by using information from the *foreign* entity header. The definitions of *local* entities are found in the current binary file by using a *local* entity offset.

Low memory footprint. Practice proves that applications make no use of most file data. Memory footprint of a file can be reduced significantly by grouping the frequently-used data. The described binary file format supports this feature by using offsets and by not specifying the required placement of entities in relation to one another.

Extensibility and Compatibility. The binary file format supports future changes via version number. The version field in the header is 4-byte long, and is encoded as byte array to avoid misinterpretation on platforms with different endianness. A tool that supports version `'N'` of the binary file format must also support the format of version `'N-1'`.

21.3.1 Alignment

Most entities have the 4-byte alignment in order to improve the speed of data access. All exceptions are specified explicitly in appropriate descriptions below.

21.3.2 Endianness

All multibyte values are little-endian as most target architectures are.

21.3.3 Offsets

Unless specified otherwise, all offsets are calculated from the beginning of the file.

An offset cannot contain values in the range `[0; sizeof(Header))`, except in certain cases specifically mentioned.

21.3.4 Data Types

Integer Types

Type	Description
`uint8_t`	8-bit unsigned integer value
`uint16_t`	16-bit unsigned integer value
`uint32_t`	32-bit little-endian unsigned integer value
`uleb128`	Unsigned integer value in leb128 encoding
`sleb128`	Signed integer value in leb128 encoding

String Format

Alignment: None

Format:

Name	Format	Description
`utf16_length`	`uleb128`	`len << 1 | is_ascii` where `len` is the length of the string in UTF-16 code units
`data`	`uint8_t[]`	`\0` terminated character sequence in MUTF-8 (Modified UTF-8) encoding

TaggedValue Format

Alignment: None

Format:

Name	Format	Description
`tag_value`	`uint8_t`	Tag that determines the encoding of `data`.
`data`	`uint8_t[]`	Optional payload

Source Language

Source language can be specified for classes and methods. Default language for methods is that of the class. A file can be emitted from the sources written in the following languages:

Source	Encoding
`Ark Assembly`	`0x01`

Values `0x00`, `0x02` - `0x0f` are reserved.

21.3.5 File Layout

A binary file begins with the *Header* located at offset `0`. Any other data can be reached from the header.

Header Format

Alignment: 4 bytes

Format:

Name	Format	Description
`magic`	`uint8_t[8]`	Magic string - 'PANDA000'.
`checksum`	`uint8_t[4]`	`adler32` checksum of the file except magic and checksum fields.
`version`	`uint8_t[4]`	Version of file format.
`file_size`	`uint32_t`	Size of the file in bytes.
`foreign_off`	`uint32_t`	Offset to a foreign region. The region must contain *ForeignField*, *ForeignMethod*, or *ForeignClass* only. `foreign_off` does not necessarily point at the first *foreign* entity. `foreign_off` and `foreign_size` must be used at runtime to determine whether a given offset is *foreign* or *local*.
`foreign_size`	`uint32_t`	Size of the foreign region in bytes.
`num_classes`	`uint32_t`	Number of classes defined in the file, and of elements in the entity pointed at by `class_idx_off`.
`class_idx_off`	`uint32_t`	Offset to the class index. The offset must point at an entity in the *ClassIndex* format.
`export_data_size`	`uint32_t`	Size of the export data region in bytes. The region elements are pointed at by `export_data_off`.
`export_data_off`	`uint32_t`	Offset to the exported data region. The offset must point at a region in the *ExportData* format.
`num_lnps`	`uint32_t`	Number of line number programs in the file, and of elements in the entity pointed at by `lnp_idx_off`.
`lnp_idx_off`	`uint32_t`	Offset to the line number program index. The offset must point to an entity in the *LineNumberProgramIndex* format.
`num_literalarrays`	`uint32_t`	Number of literalArrays defined in the file, and of elements in the entity pointed at by `literalarray_idx_off`.
`literalarray_idx_off`	`uint32_t`	Offset to the literal array index. The offset must point at an entity in the *LiteralArrayIndex* format.
`num_index_regions`	`uint32_t`	Number of the index regions in the file, and of elements in the entity pointed at by `index_section_off`.
`index_section_off`	`uint32_t`	Offset to the index section. The offset must point at an entity in the *RegionIndex* format.

Constraint: size of header must be greater than 16 bytes. *FieldType* uses this fact.

ClassIndex Format

`ClassIndex` entities are designed to allow to quickly find a type definition by name at runtime. It is organized as an array of offsets to a *Class* or to a *ForeignClass*. All offsets are sorted by the corresponding class names.

Alignment: 4 bytes

Format:

Name	Format	Description
`offsets`	`uint32_t[]`	Sorted array of offsets to a *Class* or to a *ForeignClass*.

LineNumberProgramIndex Format

`LineNumberProgramIndex` entities are designed to allow using more compact references to *Line Number Program*. It is organized as an array of offsets to a *Line Number Program*

Alignment: 4 bytes

Format:

Name	Format	Description
`offsets`	`uint32_t[]`	Array of offsets to a *Line Number Program*.

LiteralArrayIndex Format

`LiteralArrayIndex` entities are designed to allow to find a *LiteralArray* definition by index at runtime. It is organized as an array of offsets to a *LiteralArray*.

Alignment: 4 bytes

Format:

Name	Format	Description
`offsets`	`uint32_t[]`	Sorted array of offsets to a *LiteralArray*.

RegionIndex Format

`RegionIndex` entities are designed to allow finding the index that covers a specified offset in the file at runtime. It is organized as sorted array of a *RegionHeader*.

Alignment: 4 bytes

Format:

Name	Format	Description
`region_headers`	`RegionHeader[]`	File regions sorted by the start offset of the region. Each element must have the *RegionHeader* format.

Constraint: Two different regions must not overlap with each other.

RegionHeader Format

To address file entities by using 16-bit indices, a binary file is split into regions. The information on each region is stored in the `RegionHeader` format.

Alignment: 4 bytes

Format:

Name	Format	Description
`start_off`	`uint32_t`	Start offset of the region.
`end_off`	`uint32_t`	End offset of the region.
`class_idx_size`	`uint32_t`	Number of elements in an index pointed at by `class_idx_off`. Max value is 65536.
`class_idx_off`	`uint32_t`	Offset to a class index. The offset must point at a *ClassRegionIndex*.
`method_idx_size`	`uint32_t`	Number of elements in an index pointed at by `method_idx_off`. Max value is 65536.
`method_idx_off`	`uint32_t`	Offset to a method index. The offset must point at a *MethodRegionIndex*.
`field_idx_size`	`uint32_t`	Number of elements in an index pointed at by `field_idx_off`. Max value is 65536.
`field_idx_off`	`uint32_t`	Offset to a field index. The offset must point at a *FieldRegionIndex*.
`proto_idx_size`	`uint32_t`	Number of elements in an index pointed at by `proto_idx_off`. Max value is 65536.
`proto_idx_off`	`uint32_t`	Offset to a proto index. The offset must point at a *ProtoRegionIndex*.

ClassRegionIndex Format

`ClassRegionIndex` entities are designed to allow finding a type definition by index at runtime. It is organized as an array of offsets to a *Class* or to a *ForeignClass*.

Alignment: 4 bytes

Format:

Name	Format	Description
`types`	`uint32_t[]`	Offsets to a *Class* or to a *ForeignClass*.

MethodRegionIndex Format

`MethodRegionIndex` entities are designed to allow finding a method definition by index at runtime. It is organized as an array of offsets to a *Method* or to a *ForeignMethod*.

Alignment: 4 bytes

Format:

Name	Format	Description
`methods`	`uint32_t[]`	Array of offsets to a *Method* or to a *ForeignMethod*.

FieldRegionIndex Format

`FieldRegionIndex` entities are designed to allow finding a field definition by index at runtime. It is organized as an array of offsets to a *Field* or to a *ForeignField*.

Alignment: 4 bytes

Format:

Name	Format	Description
`fields`	`uint32_t[]`	Array of offsets to a *Field* or to a *ForeignField*.

ProtoRegionIndex Format

`ProtoRegionIndex` entities are designed to allow finding a proto definition by index at runtime.

Alignment: 4 bytes

Format:

Name	Format	Description
`protos`	`uint32_t[]`	Array of offsets to a *Proto*.

ExportData Format

`ExportData` region provides information about exported declarations, to be able to import the binary file from another source file and resolve to its declarations at compile-time. This information is mandatory if the binary file is distributed as a library without available sources. The information is read by the compiler front-end and used when resolving and checking references to the imported file.

It is organized as two subsections according to two existing mechanisms of import resolution to the binary file.

Alignment: None

Format:

Name	Format	Description
`metada`	`uint`	A flag of whether metadata was emitted for the file. 0 means metadata emitting was disabled, any other value means that enabled.
`metada`	`uint`	Optional. The size of metadata field in bytes. This field is absent if the `metadataEnabled` flag is set to 0.
`export`	`Clas`	A class index for the exported classes. If `metadataEnabled` is set to 0, the size of `exportTable` in bytes equals to `export_data_size - sizeof(uint32_t)`. Otherwise, the size of `exportTable` in bytes is calculated by the formula `export_data_size - sizeof(uint32_t) * 2 - metadataSize`.
`metada`	`uint`	Optional. The encoded binary data of the exported declarations. This field is absent if the `metadataEnabled` flag is set to 0.

ForeignField Format

Alignment: None

Format:

Name	Format	Description
`class_idx`	`uint16_t`	Index of a declaring class in the corresponding *ClassRegionIndex*. The corresponding index entry must be an offset to a *Class* or to a *ForeignClass*.
`type_idx`	`uint16_t`	Index of the field type in the corresponding *ClassRegionIndex*. The corresponding index entry must have the *FieldType* format.
`name_off`	`uint32_t`	Offset to the name of the field. The offset must point at a *String*.
`access_flags`	`uleb128`	Access flags of a field. The value must be a combination of *Field Access Flags*.

> **Note**
>
> A proper region index to resolve `class_idx` and `type_idx` can be found by a foreign field offset.

Field Format

Alignment: None

Format:

Name	Format	Description
`class_idx`	`uint16_t`	Index of a declaring class in the corresponding *ClassRegionIndex*. The corresponding index entry must be an offset to a *Class* or to a *ForeignClass*.
`type_idx`	`uint16_t`	Index of the field type in the corresponding *ClassRegionIndex*. The corresponding index entry must have the *FieldType* format.
`name_off`	`uint32_t`	Offset to the name of the field. The offset must point at a *String*.
`access_flags`	`uleb128`	Access flags of the field. The value must be a combination of *Field Access Flags*.
`field_data`	`TaggedValue[]`	Variable length list of tagged values. Each element must have the *TaggedValue* format. A tag must have values from *Field Tags*, and follow the increasing tag order (except `0x00` tag, which must be the last).

> **Note**
>
> A proper region index to resolve `class_idx` and `type_idx` can be found by a foreign field offset.

FieldType

Any offset in the range `[0; sizeof(Header))` is invalid because the first bytes of the file contain the header. The `FieldType` encoding uses this fact to encode primitive types of the field in the low 4 bits. The value of a non-primitive type is an offset to a *Class* or to a *ForeignClass*. In either case, *FieldType* is `uint32_t`. Primitive types are encoded as follows:

Type	Encoding
`u1`	`0x00`
`i8`	`0x01`
`u8`	`0x02`
`i16`	`0x03`
`u16`	`0x04`
`i32`	`0x05`
`u32`	`0x06`
`f32`	`0x07`
`f64`	`0x08`
`i64`	`0x09`
`u64`	`0x0a`
`any`	`0x0b`

Field Access Flags

Name	Format	Description
ACC_PUBLIC	0x0001	Declared public.
ACC_PRIVATE	0x0002	Declared private.
ACC_PROTECTED	0x0004	Declared protected.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final.
ACC_READONLY	0x0020	Declared readonly.
ACC_VOLATILE	0x0040	Declared volatile.
ACC_TRANSIENT	0x0080	Declared transient.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ENUM	0x4000	Declared an element of an enum.

Constraint: Only the ACC_STATIC flag is used for a *ForeignField*. Other flags are ignored.

Field Tags

Name	Tag	Quantity	Data Format	Description
NOTHING	0x00	1	none	No more values. The value with this tag must be the last.
INT_VALUE	0x01	0-1	sleb128	Integral value of a field. This tag is used when a field has type boolean, byte, char, short or int.
VALUE	0x02	0-1	uint8_t[4]	Contains a *Value*. Encoding depends on type_idx.
RUNTIME_ANNOTATIONS	0x03	>=0	uint8_t[4]	Offset to a field annotation **visible** at runtime. If a field has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
ANNOTATIONS	0x04	>=0	uint8_t[4]	Offset to a field annotation **invisible** at runtime. If a field has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
RUNTIME_TYPE_ANNOTATION	0x05	>=0	uint8_t[4]	Offset to a field type annotation **visible** at runtime. If a field has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
TYPE_ANNOTATION	0x06	>=0	uint8_t[4]	Offset to a field type annotation **invisible** at runtime. If a field has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.

Constraint: INT_VALUE and VALUE tags are mutually exclusive.

ForeignMethod Format

Alignment: None

Format:

Name	Format	Description
`class_idx`	`uint16_t`	Index of a declaring class in the corresponding *ClassRegionIndex*. The corresponding index entry must be an offset to a *Class* or to a *ForeignClass*.
`proto_idx`	`uint16_t`	Index of a method prototype in the corresponding *ProtoRegionIndex*. The corresponding index entry must be an offset to a *Proto*.
`name_off`	`uint32_t`	Offset to the name of the method. The offset must point at a *String*.
`access_flags`	`uleb128`	Access flags of the method. The value must be a combination of *Method Access Flags*.

Note

A proper region index to resolve `class_idx` and `proto_idx` can be found by a foreign method offset.

Method Format

Alignment: None

Format:

Name	Format	Description
`class_idx`	`uint16_t`	Index of a declaring class in the corresponding *ClassRegionIndex*. The corresponding index entry must be an offset to a *Class* or to a *ForeignClass*.
`proto_idx`	`uint16_t`	Index of a method prototype in the corresponding *ProtoRegionIndex*. The corresponding index entry must be an offset to a *Proto*.
`name_off`	`uint32_t`	Offset to the name of the method. The offset must point at a *String*.
`access_flags`	`uleb128`	Access flags of the method. The value must be a combination of *Method Access Flags*.
`method_data`	`TaggedValue[]`	Variable length list of tagged values. Each element must have the *TaggedValue* format. A tag must have values from *Method Tags*, and follow the increasing tag order (except `0x00` tag, which must be the last).

Note

A proper region index to resolve `class_idx` and `proto_idx` can be found by a foreign method offset.

Proto Format

Alignment: 2 bytes

Format:

Name	Format	Description
shorty	uint16_t[]	Short representation of a prototype. Syntax is described in *Shorty*.
reference_types	uint16_t[]	Array of indices of the non-primitive types in a method signature. Each ref type in a shorty has a corresponding element in the array. The size of the array equals the number of ref types in the shorty.

Note

A proper region index to resolve reference type indices can be found by the proto offset.

Shorty Syntax

Shorty is a short description of a method signature without detailed information on reference types. A shorty begins with a return type followed by method arguments, and ends with 0x0.

The shorty syntax is as follows:

```
Shorty:
    ReturnType ParamTypeList 0x0
    ;
ReturnType:
    Type
    ;
ParamTypeList:
    (Type)*
    ;
```

Type encoding:

Type	Encoding
void	0x01
u1	0x02
i8	0x03
u8	0x04
i16	0x05
u16	0x06
i32	0x07
u32	0x08
f32	0x09
f64	0x0a
i64	0x0b
u64	0x0c
any	0x0d
ref	0x0e

All Shorty elements are divided into groups of four elements starting from the beginning. Each group is encoded in uint16_t. Each element is encoded in 4 bits. A group of four elements v1, v2, v3, and v4 is encoded in uint16_t as follows:

Values	Bits
v1	0-3
v2	4-7
v3	8-11
v4	12-15

If a group contains less than 4 elements, then the rest bits are filled with `0x0`.

Method Access Flags

Name	Format	Description
ACC_PUBLIC	0x0001	Declared public.
ACC_PRIVATE	0x0002	Declared private.
ACC_PROTECTED	0x0004	Declared protected.
ACC_STATIC	0x0008	Declared static.
ACC_FINAL	0x0010	Declared final.
ACC_SYNCHRONIZED	0x0020	Declared synchronized.
ACC_BRIDGE	0x0040	Bridge method, generated by the compiler.
ACC_VARARGS	0x0080	Declared with variable number of arguments.
ACC_NATIVE	0x0100	Declared native.
ACC_ABSTRACT	0x0400	Declared abstract.
ACC_STRICT	0x0800	Declared strictfp.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.

Constraint: Only the `ACC_STATIC` flag is used for a *ForeignMethod*. Other flags are ignored.

Method Tags

Name	Tag	Quantity	Data Format	Description
NOTHING	0x00	1	none	No more values. The value with this tag must be the last.
CODE	0x01	0-1	uint8_t[4]	Offset to the code of a method. The offset must point at a *Code*.
SOURCE_LANG	0x02	0-1	uint8_t	The *Source Language* of a method.
RUNTIME_ANNOTATION	0x03	>=0	uint8_t[4]	Offset to a method annotation **visible** at runtime. If a method has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
RUNTIME_PARAM_ANNOTATION	0x04	0-1	uint8_t[4]	Offset to a method parameters annotation **visible** at runtime. The offset must point at a *ParamAnnotations*.
DEBUG_INFO	0x05	0-1	uint8_t[4]	Offset to debug information related to a method. The offset must point at a *InstDebugInfo*.
ANNOTATION	0x06	>=0	uint8_t[4]	Offset a method annotation **invisible** at runtime. If a method has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
PARAM_ANNOTATION	0x07	0-1	uint8_t[4]	Offset to a method parameters annotation **invisible** at runtime. The offset must point at a *ParamAnnotations*.
TYPE_ANNOTATION	0x08	>=0	uint8_t[4]	Offset a type annotation of a method **invisible** at runtime. If a method has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
RUNTIME_TYPE_ANNOTATION	0x09	>=0	uint8_t[4]	Offset a method annotation **visible** at runtime. If a method has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
PROFILE_INFO	0x0a	>=0	uint8_t[]	Profile information. Format is unspecified and determined by the language used in a file.

Code Format

Alignment: None

Format:

Name	Format	Description
num_vregs	uleb128	Number of registers (without argument registers).
num_args	uleb128	Number of arguments.
code_size	uleb128	Size of instructions in bytes.
tries_size	uleb128	Number of try blocks.
instructions	uint8_t[]	Instructions.
try_blocks	TryBlock[]	Array of try blocks. The array has tries_size elements. Each element has the *TryBlock* format.

TryBlock Format

Alignment: None

Format:

Name	Format	Description
start_pc	uleb128	Start pc of a TryBlock. This pc points to the first instruction covered by a TryBlock.
length	uleb128	Number of instructions covered by a TryBlock.
num_catches	uleb128	Number of catch blocks associated with a TryBlock.
catch_blocks	CatchBlock[]	Array of *CatchBlock* associated with a TryBlock. The array has num_catches elements in *CatchBlock* format. Catch blocks are ordered in the order in which their exception type must be checked at runtime. The *catch all* block, if present, must be the last.

CatchBlock Format

Alignment: None

Format:

Name	Format	Description
type_idx	uleb128	Index + 1 of the exception type handled by a *CatchBlock*, or 0 in case of a *catch all* block. The corresponding index entry must be an offset to a *ForeignClass* or to a *Class*. If the index is 0, then it is a *catch all* block that catches all exceptions.
handler_pc	uleb128	pc of the first instruction of the exception handler.
code_size	uleb128	Handler code size in bytes.

Note

A proper region index to resolve type_idx can be found by the corresponding method offset.

ParamAnnotations Format

Alignment: 8 bytes

Format:

Name	Format	Description
`count`	`uint32_t`	Number of parameters that a method has. This number includes synthetic and mandated parameters.
`annotations`	`AnnotationArray[]`	Array of annotation lists for each parameter. The array has `count` elements. Each element must have the *AnnotationArray* format.

AnnotationArray Format

Alignment: 8 bytes

Format:

Name	Format	Description
`count`	`uint32_t`	Number of elements in the array.
`offsets`	`uint32_t[]`	Array of offsets to parameter annotations. Each offset must refer to an *Annotation*. The array has `count` elements.

ForeignClass Format

Alignment: None

Format:

Name	Format	Description
`name`	`String`	Name of a foreign type. The name must conform to *Type Descriptor* syntax.

Class Format

Alignment: None

Format:

Name	Format	Description
`name`	`String`	Name of a type. The name must conform to *Type Descriptor* syntax.
`super_class_off`	`uint8_t[4]`	Offset to the name of a super class or `0` for root object class. A non-zero offset must point at a *ForeignClass* or at a *Class*.
`access_flags`	`uleb128`	Access flags of a class. A value must be a combination of *Class Access Flags*.
`num_fields`	`uleb128`	Number of fields a class has.
`num_methods`	`uleb128`	Number of methods a class has.
`class_data`	`TaggedValue[]`	Variable length list of tagged values. Each element must have the *TaggedValue* format. A tag must have values from *Class Tags*, and follow the increasing tag order (except `0x00` tag, which must be the last).
`fields`	`Field[]`	Class fields. Number of elements is `num_fields`. Each element must have the *Field* format.
`methods`	`Method[]`	Class methods. Number of elements is `num_methods`. Each element must have the *Method* format.

Type Descriptor

Type descriptors in a binary file have the following syntax:

```
TypeDescriptor:
    PrimitiveType
    | ArrayType
    | RefType
    | UnionType
    ;
PrimitiveType:
    'Z'
    | 'B'
    | 'H'
    | 'S'
    | 'C'
    | 'I'
    | 'U'
    | 'F'
    | 'D'
    | 'J'
    | 'Q'
    | 'A'
    ;
ArrayType:
    '[' TypeDescriptor
    ;
RefType:
    'L' RefTypeName ';'
    ;
UnionType:
    '{U' TypeDescriptor TypeDescriptor (TypeDescriptor)* '}'
    ;
```

`RefTypeName` is a *Fully Qualified Name* of the type with all dots (`'.'`) replaced for slashes (`'/'`).

`PrimitiveType` encoding:

Type	Encoding	Description
`u1`	`Z`	1-bit unsigned integer
`i8`	`B`	8-bit signed integer
`u8`	`H`	8-bit unsigned integer
`i16`	`S`	16-bit signed integer
`u16`	`C`	16-bit unsigned integer
`i32`	`I`	32-bit signed integer
`u32`	`U`	32-bit unsigned integer
`f32`	`F`	Floating-point number with single precision
`f64`	`D`	Floating-point number with double precision
`i64`	`J`	64-bit signed integer
`u64`	`Q`	64-bit unsigned integer
`any`	`A`	any

Constraint: `UnionType` must be canonicalized in a binary file to ensure that no equivalent union type is present. Canonicalization presumes alphabetically sorting `TypeDescriptor` s of all *constituent types*.

Class Access Flags

Name	Format	Description
ACC_PUBLIC	0x0001	Declared public.
ACC_FINAL	0x0010	Declared final.
ACC_SUPER	0x0020	No special meaning. Added for compatibility only
ACC_INTERFACE	0x0200	Is an interface.
ACC_ABSTRACT	0x0400	Declared abstract.
ACC_SYNTHETIC	0x1000	Declared synthetic; not present in the source code.
ACC_ANNOTATION	0x2000	Declared an annotation type.
ACC_ENUM	0x4000	Declared an enum type.

Class Tags

Name	Tag	Quantity	Data Format	Description
NOTHING	0x00	1	none	No more values. The value with this tag must be the last.
INTERFACES	0x01	0-1	uleb128 uint8_t[]	List of interfaces a class implements. Data contains the number of interfaces encoded in the uleb128 format followed by indices of interfaces in the *ClassRegionIndex* format. Each index is 2-byte long, and when resolved must point at a *ForeignClass* or at a *Class*. The number of indices is equal to the number of interfaces.
SOURCE_LANG	0x02	0-1	uint8_t	Data represents the *Source Language*.
RUNTIME_ANNOTATION	0x03	>=0	uint8_t[4]	Offset to a class annotation **visible** at runtime. If a class has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
ANNOTATION	0x04	>=0	uint8_t[4]	Offset to a class annotation **invisible** at runtime. If a class has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
RUNTIME_TYPE_ANNOTATION	0x05	>=0	uint8_t[4]	Offset to a class type annotation **visible** at runtime. If a class has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
TYPE_ANNOTATION	0x06	>=0	uint8_t[4]	Offset to a class type annotation **invisible** at runtime. If a class has several annotations, then the tag can be repeated. The offset must point at an *Annotation*.
SOURCE_FILE	0x07	0-1	uint8_t[4]	Offset to a file name string containing source code of a class.

> **Note**
>
> A proper region index to resolve interface indices can be found by the class offset.

Annotation Format

Alignment: None

Format:

Name	Format	Description
`class_idx`	`uint16_t`	Index of a declaring class in the *ClassRegionIndex* format. The corresponding index entry must be an offset to a *Class* or to a *ForeignClass*.
`count`	`uint16_t`	The number of name-value pairs in the annotation (number of elements in `elements` array).
`elements`	`AnnotationElement[]`	Array of annotation elements. Each element must have the *AnnotationElement* format. The order of elements must be the same as in the annotation class.
`element_type:`	`uint8_t[]`	Array of annotation element types. Each element in `element_types` describes the type of a corresponding *AnnotationElement* in `elements`. The order of elements in the array matches the order of the `elements` field. Each element must have a value from the *Annotation Element Type*.

> **Note**
>
> A proper region index to resolve `class_idx` can be found by the annotation offset.

Annotation Element Type

Type	Tag
`u1`	`1`
`i8`	`2`
`u8`	`3`
`i16`	`4`
`u16`	`5`
`i32`	`6`
`u32`	`7`
`i64`	`8`
`u64`	`9`
`f32`	`A`
`f64`	`B`
`string`	`C`
`record`	`D`
`method`	`E`
`enum`	`F`
`annotation`	`G`
`method_handle`	`J`

continues on next page

Table 1 – continued from previous page

Type	Tag
array	H
u1[]	K
i8[]	L
u8[]	M
i16[]	N
u16[]	O
i32[]	P
u32[]	Q
i64[]	R
u64[]	S
f32[]	T
f64[]	U
string[]	V
record[]	W
method[]	X
enum[]	Y
annotation[]	Z
method_handle[]	@
nullptr_string	*

Note

The correct value for element with `nullptr_string` tag is `0` (`\x00\x00\x00\x00`)

AnnotationElement Format

Alignment: None

Format:

Name	Format	Description
name_off	uint32_t	Offset to the element name. The offset must point at a *String*.
value	uint32_t	Value of the element. If the type of an annotation element is `boolean`, `byte`, `short`, `char`, `int`, or `float`, then the field `value` contains the value itself in the corresponding *Value* format. Otherwise, the field contains an offset to the *Value*. The format of this *Value* can be determined based on the *AnnotationElement* type.

Value Format

There are different value encodings depending on the value type.

ByteValue Format

Alignment: None

Format:

Name	Format	Description
value	uint8_t	Signed 1-byte integer value.

ShortValue Format

Alignment: 2 bytes

Format:

Name	Format	Description
value	uint8_t[2]	Signed 2-byte integer value.

IntegerValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
value	uint8_t[4]	Signed 4-byte integer value.

LongValue Format

Alignment: 8 bytes

Format:

Name	Format	Description
value	uint8_t[8]	Signed 8-byte integer value.

FloatValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
value	uint8_t[4]	4-byte bit pattern, zero-extended to the right, and interpreted as an IEEE754 32-bit floating point value.

DoubleValue Format

Alignment: 8 bytes

Format:

Name	Format	Description
value	uint8_t[8]	8-byte bit pattern, zero-extended to the right, and interpreted as an IEEE754 64-bit floating point value.

StringValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint32_t`	The value represents an offset to a *String*.

EnumValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint32_t`	The value represents an offset to an enum field. An offset must point at a *Field* or at a *ForeignField*.

ClassValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint32_t`	The value represents an offset to a *Class* or to a *ForeignClass*.

AnnotationValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint32_t`	The value represents an offset to a *Annotation*.

MethodValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint32_t`	The value represents an offset to a *Method* or to a *ForeignMethod*.

MethodHandleValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
value	uint32_t	The value represents an offset to a *MethodHandle*.

MethodTypeValue Format

Alignment: 4 bytes

Format:

Name	Format	Description
value	uint32_t	The value represents an offset to a *Proto*.

ArrayValue Format

Alignment: None

Format:

Name	Format	Description
count	uleb128	Number of elements in the array.
value	uint32_t	Unaligned array of *Value* entities. The array has count elements. Format depends on the tag_value field.

LiteralArray Format

Alignment: None

Format:

Name	Format	Description
num_literals	uint32_t	Interpreted differently for different *Literal Tags*. If literals have an ARRAY_* tag, then num_literals equals to the number of elements in an array. For other tags num_literals equals to the number of elements in literals array times 2 (since each tag can be interpreted as separate *Literal*).
literals	Literal[]	Elements of a literal array. The number of elements is num_literals. Each element must have the *Literal* format.

Literal Tags

Tag	Value	data encoding
TAGVALUE	0x00	*ByteOne*
BOOL	0x01	*ByteOne*
INTEGER	0x02	*ByteFour*
FLOAT	0x03	*ByteFour*
DOUBLE	0x04	*ByteEight*
STRING	0x05	*ByteFour*
BIGINT	0x06	*ByteEight*
METHOD	0x07	*ByteFour*
GENERATORMETHOD	0x08	*ByteFour*
ACCESSOR	0x09	*ByteOne*
METHODAFFILIATE	0x0a	*ByteTwo*
ARRAY_U1	0x0b	*ByteOne*
ARRAY_U8	0x0c	*ByteOne*
ARRAY_I8	0x0d	*ByteOne*
ARRAY_U16	0x0e	*ByteTwo*
ARRAY_I16	0x0f	*ByteTwo*
ARRAY_U32	0x10	*ByteFour*
ARRAY_I32	0x11	*ByteFour*
ARRAY_U64	0x12	*ByteEight*
ARRAY_I64	0x13	*ByteEight*
ARRAY_F32	0x14	*ByteFour*
ARRAY_F64	0x15	*ByteEight*
ARRAY_STRING	0x16	*ByteFour*
ASYNCGENERATORMETHOD	0x17	*ByteFour*
ASYNCMETHOD	0x18	*ByteFour*
LITERALARRAY	0x19	*ByteFour*
NULLVALUE	0xff	*ByteOne*

Literal Format

Alignment: None

Format:

Name	Format	Description
tag	uint8_t	Tag of a literal. A tag of a literal instructs how to interpret num_literals and data fields. A tag of a literal must be one of *Literal Tags*.
data	uint8_t[]	Elements of a literal array. The number of elements is num_literals. Different data encodings depend on the value of a tag.

ByteOne Format

Alignment: None

Format:

Name	Format	Description
value	uint8_t	1-byte value.

ByteTwo Format

Alignment: 2 bytes

Format:

Name	Format	Description
`value`	`uint8_t[2]`	2-byte value.

ByteFour Format

Alignment: 4 bytes

Format:

Name	Format	Description
`value`	`uint8_t[4]`	4-byte value.

ByteEight Format

Alignment: 8 bytes

Format:

Name	Format	Description
`value`	`uint8_t[8]`	8-byte value.

InstDebugInfo Format

Debug information contains mapping between program counter of a method, line numbers in source code, and information about local variables. The format is derived from the DWARF Debugging Information Format, Version 3 (see subsection 6.2). The mapping and local variable information are encoded in the line number program, which is interpreted by the *State Machine*. All constants the program refers to are moved into the *Constant Pool* in order to deduplicate identical line number programs of different methods.

Alignment: None

Format:

Name	Format	Description
`line_start`	`uleb128`	The initial value of line register of the *State Machine*.
`num_parameters`	`uleb128`	Number of method parameters.
`parameters`	`uleb128[]`	Parameter names of a method. An array has `num_parameters` elements. Each element is an offset to a *String* or `0` if no name is available.
`constant_pool_size`	`uleb128`	Size of constant pool in bytes.
`constant_pool`	`uleb128[]`	*Constant pool* data. Length is `constant_pool_size` bytes.
`line_number_program_idx`	`uleb128`	Line number program index in the *LineNumberProgramIndex* format. Programs have variable lengths, and end in `DBG_END_SEQUENCE` opcode.

Constant Pool

Many methods have similar line number programs that differ in variable names, variable types, and file names only. In order to deduplicate such programs, all constants the program refers to are stored in the constant pool. During program interpretation, the *State Machine* tracks a pointer to the constant pool. When the *State Machine* interprets an instruction that requires a constant argument, the machine reads memory for the value pointed to by the constant pool pointer, and then increments the pointer. Thus, a program has no explicit reference to constants, and can be deduplicated.

State Machine

The aim of the state machine is to generate mapping between the program counter, line numbers, and local variable information. The machine has the following registers:

Name	Initial value	Description
`address`	0	Program counter (refers to method instructions). Must always increase monotonically.
`line`	`line_start` from *InstDebugInfo*.	Unsigned integer which corresponds to a line number in source code. All lines are numbered beginning at 1, and the register must not have a value less than 1.
`file`	Value of `SOURCE_FILE` tag in `class_data` (see *Class*) or `0`.	Offset to the name of a source file. If no such information is available (i.e., a `SOURCE_FILE` tag is absent in *Class*), then the register value is `0`.
`prologue_end`	`false`	Register indicates the current address where an entry breakpoint can be set for a method.
`epilogue_begin`	`false`	Register indicates the current address where an exit breakpoint can be set for a method.
`constant_pool_ptr`	Address of the first byte of `constant_pool` i *InstDebugInfo*.	Pointer to the current constant value.

Line Number Program Format

A line number program consists of instructions. Each instruction has one byte opcode and optional arguments. Depending on opcode, an argument value can be encoded into the instruction. Otherwise, the instruction requires the value to be read from the constant pool.

Opcode	Value	Instruction Format	Constant Pool Args	Description
END_SEQUENCE	0x00			Marks the end of line number program.
ADVANCE_PC	0x01		uleb128	Increments address register by the value that constant_pool_ptr refers to without emitting a line.
ADVANCE_LINE	0x02		sleb128	Increments line register by the value that constant_pool_ptr refers to without emitting a line.
START_LOCAL	0x03	sleb128	uleb128 uleb128	Introduces a local variable with the name and type that constant_pool_ptr refers to at the current address. The number of a register that contains the variable is encoded in the instruction. Register value -1 signifies the accumulator register. The name is an offset to a *String*, and the type is an offset to a *ForeignClass* or to a *Class*. If corresponding information is missing, then the offset is 0.
START_LOCAL_EXTENDED	0x04	sleb128	uleb128 uleb128 uleb128	Introduces a local variable with the name, type, and type signature that constant_pool_ptr refers to at the current address. The number of a register that contains the variable is encoded in the instruction. Register value -1 signifies the accumulator register. The name is an offset to a *String*, and the type is an offset to a *ForeignClass* or to a *Class*. If corresponding information is missing, then the offset is 0.
END_LOCAL	0x05	sleb128		Marks that local variable in a specified register is out of scope. The register number is encoded in the instruction. Register value -1 signifies the accumulator register.
RESTART_LOCAL	0x06	sleb128		Re-introduces a local variable at a specified register. The name and type are the same as the last local in the register. The number of the register is encoded in the instruction. Register value -1 signifies the accumulator register.
SET_PROLOGUE_END	0x07			Sets prologue_end register to true. Any special opcodes clear prologue_end register.
SET_EPILOGUE_BEGIN	0x08			Sets epilogue_end register to true. Any special opcodes clear epilogue_end register.
SET_FILE	0x09		uleb128	Sets file register to the value constant_pool_ptr refers to. The argument is an offset to a *String* which represents the file name or 0.
SET_SOURCE_CODE	0x0a		uleb128	Sets source_code register to the value constant_pool_ptr refers to. The argument is an offset to a *String* which represents the source code or 0.
SET_COLUMN	0x0b		uleb128	Sets column register by the value that constant_pool_ptr refers to
Special opcodes	0x0c . . 0xff			

Special opcodes:

State Machine interprets each special opcode as follows (see DWARF Debugging Information Format subsection 6.2.5.1 Special Opcodes):

1. Calculate the adjusted opcode: `adjusted_opcode = opcode - OPCODE_BASE`.
2. Increment `address` register: `address += adjusted_opcode / LINE_RANGE`.
3. Increment `line` register: `line += LINE_BASE + (adjusted_opcode % LINE_RANGE)`.
4. Emit line number.
5. Set `prologue_end` register to `false`.
6. Set `epilogue_begin` register to `false`.

Where:

- `OPCODE_BASE = 0x0c`: the first special opcode.
- `LINE_BASE = -4`: the smallest line number increment.
- `LINE_RANGE = 15`: the number of line increments presented.

MethodHandle Format

Alignment: none

Format:

Name	Format	Description
`type`	`uint8_t`	Type of a handle. Must be one of *Type Of MethodHandle*.
`offset`	`uleb128`	Offset to an entity of a corresponding type. Type of an entity is determined depending on the handle type (see *Type of MethodHandle*).

Type of MethodHandle

The types available for a method handle are as follows:

Name	Format	Description
PUT_STATIC	0x00	Method handle refers to a static setter. Offset in *MethodHandle* must point at a *Field* or at a *ForeignField*.
GET_STATIC	0x01	Method handle refers to a static getter. Offset in *MethodHandle* must point at a *Field* or at a *ForeignField*.
PUT_INSTANCE	0x02	Method handle refers to an instance getter. Offset in *MethodHandle* must point at a *Field* or at a *ForeignField*.
GET_INSTANCE	0x03	Method handle refers to an instance setter. Offset in *MethodHandle* must point at a *Field* or at a *ForeignField*.
INVOKE_STATIC	0x04	Method handle refers to a static method. Offset in *MethodHandle* must point at a *Method* or at a *ForeignMethod*.
INVOKE_INSTANCE	0x05	Method handle refers to an instance method. Offset in *MethodHandle* must point at a *Method* or at a *ForeignMethod*.
INVOKE_CONSTRUCTC	0x06	Method handle refers to a constructor. Offset in *MethodHandle* must point at a *Method* or at a *ForeignMethod*.
INVOKE_DIRECT	0x07	Method handle refers to a direct method. Offset in *MethodHandle* must point at a *Method* or at a *ForeignMethod*.
INVOKE_INTERFACE	0x08	Method handle refers to an interface method. Offset in *MethodHandle* must point at a *Method* or at a *ForeignMethod*.

Argument Types

A bootstrap method can accept static arguments of the following types:

Name	Format	Description
Integer	0x00	Corresponding argument has the *IntegerValue* encoding.
Long	0x01	Corresponding argument has the *LongValue* encoding.
Float	0x02	Corresponding argument has the *FloatValue* encoding.
Double	0x03	Corresponding argument has the *DoubleValue* encoding.
String	0x04	Corresponding argument has the *StringValue* encoding.
Class	0x05	Corresponding argument has the *ClassValue* encoding.
MethodHandle	0x06	Corresponding argument has the *MethodHandleValue* encoding.
MethodType	0x07	Corresponding argument has the *MethodTypeValue* encoding.

21.4 Runtime Type System

The ArkTS runtime operates upon the following types:

- *Primitive types* : `void`, `u1`, `i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `f32`, `f64`, `i64`, `u64`, `any`.
- *Reference types* : All other types. All *reference types* have corresponding *ForeignClass* or *Class* in a binary file.

The ArkTS runtime additionally distinguishes between the above *reference type* groups as follows:

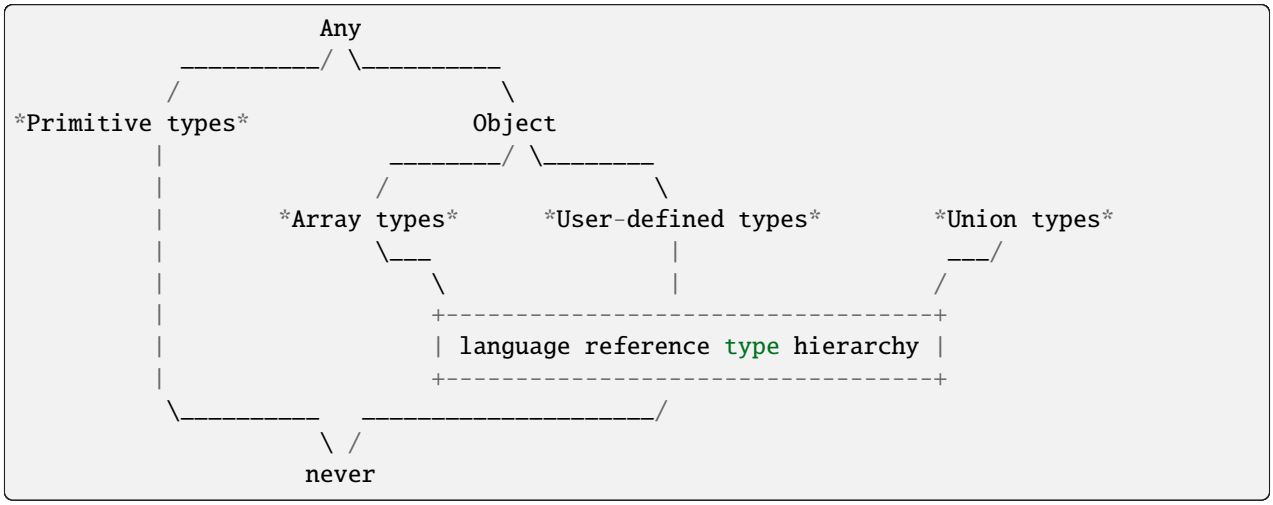
- `Any` : Top type of the runtime type system. Correspondent to the *Class* with the *Type Descriptor* that matches `RefType` with a *Fully Qualified Name* `Y`. **NOTE:** NOT the same as `any` in *primitive types*.
- `never` : Bottom type of the runtime type system. Correspondent to the *Class* with the *Type Descriptor* that matches `RefType` with a *Fully Qualified Name* `N`.
- *Array types* : Types that represent arrays of some *component type*. Correspondent to a *Class* with the *Type Descriptor* that matches `ArrayType`.

- *Union types* : Types that represent unions of their *component types*. Correspondent to a *Class* with the *Type Descriptor* that matches `UnionType`.
- `Object` : Base type of all *User-defined types* and *Array types*. Correspondent to the *Class* with the *Type Descriptor* that matches `RefType` with a *Fully Qualified Name* `std.core.Object`.
- *User-defined types* : *Reference types* that belong to none of the groups above.

Each of these types is represented at runtime with instances of the structure named `Class`. Each unique type has only a single instance of `Class` structure that falls into one of the categories above.

21.4.1 Subtyping

Reference types and *Primitive types* form the type hierarchy described below:



The following statement holds true for *primitive type* subtyping: one *primitive type* is a subtype of another *primitive type* only if both are of the same type.

For *Array types*, *Union types* and *User-defined types* subtyping is mandated by the ArkTS subtyping rules.

21.5 Runtime Name

The ArkTS runtime (e.g. standard library reflection, and class loading APIs) and build time (build system, compiler, bytecode manipulation tools) use *runtime name* to work with modules, classes, and other entities. The syntax of a *runtime name* is presented below:

```
RuntimeName:
    PrimitiveType
    | ArrayType
    | RefType
    | UnionType
    ;
PrimitiveType:
    "void"
    | "u1"
```

(continues on next page)

(continued from previous page)

```
    | "i8"
    | "u8"
    | "i16"
    | "u16"
    | "i32"
    | "u32"
    | "f32"
    | "f64"
    | "i64"
    | "u64"
    | "any"
    ;
ArrayType:
    RuntimeName '[]'
    ;
RefType:
    RefTypeName '[]'
    ;
UnionType:
    '{U' RuntimeName ',' RuntimeName (',' RuntimeName)* '}'
    ;
```

`RefTypeName` is the *Fully Qualified Name* of the type.

Constraint: `UnionType` must be canonicalized. Canonicalization presumes sorting `RuntimeName` s of all *constituent types* alphabetically.

21.6 Language Representation in Binary File

This chapter describes how the ArkTS compiler translates different language types and constructs into bytecode.

> **Note**
>
> The ArkTS *type erasure* rules take precedence over the translation rules discussed in this chapter. It means that the *type erasure* rules are applied first for any ArkTS type `T`, and then the *effective type* of `T` is translated into the binary representation in accordance with the rules below.

21.6.1 Value Types

The ArkTS *value types* can be represented in a binary file in **one** of the following two forms depending on the context:

- `PrimitiveType` in *Type Descriptor*, *FieldType*, or *Shorty*; or
- Corresponding predefined *reference type*.

The *Type Descriptor* of a *reference type* type matches `RefType`. Mapping between the ArkTS type, *primitive type* and *Fully Qualified Name* of a *reference type* is presented in the table below:

Type	Primitive	Reference
boolean	u1	std.core.Boolean
char	u16	std.core.Char
byte	i8	std.core.Byte
short	i16	std.core.Short
int	i32	std.core.Int
long	i64	std.core.Long
float	f32	std.core.Float
number,double	f64	std.core.Double

Selecting Between Primitive And Reference

Each time one of *value types* is used, the ArkTS compiler tries to lower it to its corresponding *primitive type* representation by default. However, *reference type* representation is required in some cases. A comprehensive list of such scenarios is presented below:

- *Component types* of a *union type* must be *reference types*, i.e., any *value type* in a *union type* is represented as a *reference type* in a binary file.
- If a *value type* is used as a constraint of a generic parameter, then it is also represented as a *reference type* in a binary file.
- If a default value for a parameter of a method or a function is of a *value type*, then the type of that parameter is represented with *reference type* in a binary file.
- When a field of a class is inherited as an instantiation of a field of a generic class whose type is a type parameter, that field is represented with *reference* type.
- If a method is inherited from a generic class or interface without overriding, its parameters whose type in the original definition is a type parameter are represented with *reference* type. The same applies to the return type of a method. If, on the other hand, such method is overridden, the paremeters and return type are represented as *primitives*.
- The return type of a `valueOf` methods for *value types* is represented as *reference*.
- In all other cases, *primitive* representation is selected.

```
abstract class B<T> {
   n: number = 0.
   x: T

   f(i: number, y: T): T { return y }
   g(y: T): T { return y}
}
class D extends B<number> {
   override g: number = 1.
   override g(y: number) { return y }
}
let d = new D
d.n                       // represented as primitive
d.x                       // represented as reference
let v1 = d.f(2., 3.)      // 2. is represented as primitive, 3. as reference
                          // function return is a reference,
                          // unboxed before assigning to v1
let v2 = d.g(4.)          // 4. is represented as primitive,
```

(continues on next page)

(continued from previous page)

```
                            // returned as primitive
let v3 = (d as B).g(5.)     // 5. is represented as reference,
                            // return value is reference,
                            // unboxed before assigning to v3
```

21.6.2 Type `Any`

The ArkTS type `Any` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `Y`.

21.6.3 Type `Object`

The ArkTS type `Object` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.Object`.

21.6.4 Type `never`

The ArkTS type `never` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `N`.

21.6.5 Type `void`

The ArkTS type `void` is represented in a binary file as the primitive type `void` in the *Shorty*.

21.6.6 Type `null`

The ArkTS type `null` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.Null`.

21.6.7 Type `string`

The ArkTS type `string` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.String`.

21.6.8 Type `bigint`

The ArkTS type `bigint` is represented in a binary file with the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.BigInt`.

21.6.9 Array types

Resizable Array Types

The ArkTS *resizable array types* are represented in a binary file by the predefined *Class*. The *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.Array`.

Fixed-Size Array Types

The ArkTS *fixed-size array types* are represented in a binary file by a predefined *Class*. Each distinct *fixed-size array type* has a unique correspondent predefined *Class*. The *Type Descriptor* of this *Class* matches `ArrayType`. *Fully Qualified Name* of the component type and of the *component type* of a *fixed-size array type* are the same.

21.6.10 Tuple Types

The ArkTS *tuple types* are represented in a binary file by a predefined *Class*. *Type Descriptor* of this *Class* matches `RefType` with *Fully Qualified Name* `std.core.Tuple1`, `std.core.Tuple2`, ..., `std.core.Tuple16`, or `std.core.TupleN` that depends on the number of constituent types of a *tuple type*.

Constraint: As shown above, the ArkTS runtime does not distinguish between tuple types with a number of constituent types greater than `16`.

21.6.11 Functional Types

The ArkTS *functional types* are represented in a binary file by a predefined *Class* that depends on the number of parameters of the *functional type* and on the presence of a *rest parameter* in the signature. The *Type Descriptor* of this *Class* for *functional types* without a *rest parameter* matches `RefType` with the *Fully Qualified Name* `std.core.Function0`, `std.core.Function1`, ..., `std.core.Function16`, or `std.core.FunctionN`. The exact *Type Descriptor* depends on the number of parameters of the *functional type*. However, the *Type Descriptor* of this *Class* for *functional types* with a *rest parameter* matches `RefType` with the *Fully Qualified Name* `std.core.FunctionR0`,

`std.core.FunctionR1`, ..., `std.core.FunctionR15`, or `std.core.FunctionR16`. The exact *Type Descriptor* depends on the number of parameters of the *functional type*.

Functional Objects

The ArkTS *functional objects* are represented in a binary file by a predefined *Class*. Each *functional object* has a unique correspondent predefined *Class*. `fields` of this *Class* contain references captured by the *functional object*. `methods` of this *Class* contain auxiliary functions needed to invoke the *functional object*. The *Method* that contains the body of the *functional object* is added to the `methods` of the enclosing class of the *functional object*. It is to allow the body of the *functional object* to access private members of the enclosing class.

21.6.12 Union Types

The ArkTS *union types* are represented in a binary file by a predefined *Class*. Each *union type* has a unique correspondent predefined *Class*. *Type Descriptor* of this *Class* matches `UnionType` with *qualified names* of the *component types* being the *qualified names* of the *component types* of the *union type*.

21.6.13 Utility Types

Awaited

The ArkTS type `Awaited` is fully expanded at compile time, and does not appear at runtime.

NonNullable

The ArkTS type `NonNullable` is fully expanded at compile time, and does not appear at runtime.

Partial

The ArkTS type `Partial` is represented in a binary file by a predefined *Class*. Each `Partial` type has a unique correspondent predefined *Class*. *Type Descriptor* of this *Class* matches `RefType` with the *unqualified name* `%%partial-typeName`, where `typeName` is an *unqualified name* of the `Partial`'s type argument.

Required

The ArkTS type `Required` is fully expanded at compile time, and does not appear at runtime.

Readonly

The ArkTS type `Readonly` is fully expanded at compile time, and does not appear at runtime.

Record

The ArkTS type `Record` is represented in a binary file by a predefined *Class*. *Type Descriptor* of this *Class* matches `RefType` with *qualified name* `std.core.Record`.

ReturnType

The ArkTS type `ReturnType` is fully expanded at compile time, and does not appear at runtime.

21.6.14 Class Types

Class Declaration

Each *class declaration* is lowered to a *Class* with an *unqualified name* equal to the *class* name in the source code. `fields` of this *Class* correspond to all *class field declarations* of the *class*. `methods` of this *Class* correspond to all *class method declarations* and *class accessor declarations* of the *class*. Additionally, for each *constructor declaration* of the *class* a method is generated with the name `<ctor>` and for all *static block* in *class* the single *static method* is generated with the name `<cctor>`.

A *Method* with the name `<ctor>` is generated for each *constructor declaration* of the *class*. A *static* (`access_flags | ACC_STATIC == 1`) *Method* with the name `<cctor>` is generated for each *static block* in the *class*.

Class Extension Clause

Direct superclass of the *class* is stored in the `super_class_off` field of the *Class*.

Class Implementation Clause

Direct superinterfaces of the *class* stored in the field `class_data` of the *Class*. The tag `INTERFACES` is used to store *direct superinterfaces*.

Class Field

Each *class field declaration* is lowered to a *Field* with the name equal to the *field* name in the source code. Access modifiers of the *field* are represented by the corresponding *Field Access Flags*:

Modifier	Access Flag
`private`	`ACC_PRIVATE`
`public`	`ACC_PUBLIC`
`protected`	`ACC_PROTECTED`
`static`	`ACC_STATIC`
`readonly`	`ACC_READONLY`

Class Method

Each *class method declaration* is lowered to a *Method* with the name equal to the *method* name in the source code. Modifiers of the *method* are represented by the corresponding *Method Access Flags*:

Modifier	Access Flag
`private`	`ACC_PRIVATE`
`public`	`ACC_PUBLIC`
`protected`	`ACC_PROTECTED`
`abstract`	`ACC_ABSTRACT`
`final`	`ACC_FINAL`
`native`	`ACC_NATIVE`
`static`	`ACC_STATIC`

Class Accessor

Each *class accessor declaration* is lowered to a *Method*. The *Method* name of a getter for the property `propName` is `%%get-propName`. The *Method* name of a setter for the property `propName` is `%%set-propName`.

21.6.15 Interface Types

Interface Declaration

Each *interface declaration* is lowered to a *Class* with an *unqualified name* equal to the *interface* name in the source code. This *Class* is an *interface* (`access_flags | ACC_INTERFACE == 1`, `access_flags | ACC_ABSTRACT == 1`). `fields` of this *Class* must be empty. `methods` of this *Class* correspond to all *interface properties* and *interface method declarations* of the *interface*.

Superinterfaces And Subinterfaces

The representation of *direct superinterfaces* of an *interface* and of *Class Implementation Clause* is the same.

Interface Property

The representation of an interface accessor and of the *Class Accessor* is the same.

Interface Method

Each *interface method declaration* is lowered to a *Method* with the name equal to the *method* name in the source code. This *Method* is an *abstract method*.

21.6.16 Enumeration Types

Each *enumeration declaration* is lowered to a *Class* with an *unqualified name* equal to the *enumeration type* name in the source code. `fields` of this *Class* correspond to all *enum constants* of the *enumeration type*. `methods` of this class correspond to the *enumeration methods* of the *enumeration type*. A single *static method* with the name `<cctor>` is generated additionally for this *Class*. This *static method* corresponds to the *enumeration type* static constructor. This *Class* extends the *Class* with *Fully Qualified Name* `std.core.BaseEnum`.

21.6.17 Namespaces And Modules

Each *namespace declaration* is lowered to a *Class* with an *unqualified name* equal to the *namespace* name in the source code. This *Class* is *abstract* (`access_flags | ACC_ABSTRACT == 1`), and has an *Annotation* with the *Fully Qualified Name* `ets.annotation.Module`. `fields` of this *Class* correspond to all *variable declarations* and *constant declarations* of the *namespace*. `methods` of this *Class* correspond to all *function declarations*, *accessor declarations*, *function with receiver declarations*, and *accessor with receiver declarations* of the *namespace*. Additionally, each *namespace* has a generated *static method* with the name `<cctor>`. It corresponds to the static initializer of the *namespace*. A name for *top-level* namespace is chosen by the *build system*, while the binary representation remains.

21.6.18 Annotations

Annotation Declaration

Each *annotation declaration* is lowered to a *Class* with an *unqualified name* equal to the *annotation* name in the source code. This *Class* is an *annotation* (`access_flags | ACC_ANNOTATION == 1`, `access_flags | ACC_ABSTRACT == 1`). `fields` of this *Class* correspond to all *annotation fields* of the *annotation*. `methods` of this *Class* must be empty.

Annotation Field

The representation of each *annotation field* and *Class field* in bytecode is the same.

21.7 Verification

Even though the ArkTS compiler ensures that user program is structurally correct and does not violate ArkTS type system, and it produces only binary files that satisfy all the static and structural constraints of the *Binary File Format*, the ArkTS runtime has no guarantee that the loaded binary file was generated by such compiler or is properly formed.

> **Note**
>
> An additional problem with compile-time checking is the potential conflict between different versions of the binary files. Suppose that the module `A` that depends on the module `B` was successfully compiled. Since the time `A` was compiled, the definition of the module `B` might have changed in a way that breaks the binary compatibility (see *Binary Compatibility*). Such conflict can only be detected at runtime by the process of verification.

Because of the aforementioned problems, ArkTS runtime requires all loaded binary files to be verified, and rejects all non-verified binary files with a runtime error.

Verification ensures that the binary representation of the program is structurally correct and that the ArkTS type system is not violated in any way. For example, verification checks that no final class is later extended; that called method parameter types and count matches this method's definition; and that none of the instructions violate the ArkTS type system.

If an error occurs during verification, then an error will be thrown at the point in the program that caused the verification to start.

21.7.1 Verification Process

Depending on the implementation, the verification process may happen:

- At runtime - on *Class* loading.
- Statically - if a program linkage sequence is known prior to it's execution.

For both of these options the verification consists of these main steps:

- Verification of the dependent entities (see *Dependent Entity Verification*).
- Verification of the type system (see *Type System Verification*)
- Verification of the control flow of the program (see *Control Flow Verification*).
- Abstract interpretation of the program (see *Abstract Interpretation*).

For the *Dependent Entity Verification* and the *Type System Verification*. the unit of verification is a *Class*. For the *Abstract Interpretation* and the *Control Flow Verification*, the unit of verification is a *Method*.

Note

The code for each *Method* is verified independently of each other.

21.7.2 Dependent Entity Verification

This step verifies that all dependent entities of the *Class* can be resolved an produce no error on verification. If any of the dependent entities can not be resolved or fails the verification, an error is thrown.

21.7.3 Type System Verification

This step verifies that the *Class* declaration does not violate the existing static type system that consists of both ArkTS types and other user-defined types. This step includes, but is not limited to the following checks:

- Check that the *Class* does not extend *final Class*.
- Check that the *Class* has access to the *Class* it extends.
- etc.

If any of the checks performed at this step fail, an error is thrown.

21.7.4 Control Flow Verification

This step verifies that the control flow of the *Method* is consistent. This step includes, but is not limited to the following checks:

- The targets of all control flow instructions are each the start of an instruction.
- Control flow of a *Method* can not reach outside of it.
- Control flow can only reach the body of an exception handler via dedicated instructions.
- Control flow of a *Method* end must terminate with any of the proper instructions.
- etc.

If any of the checks performed at this step fail, an error is thrown.

21.7.5 Abstract Interpretation

This step checks the *Method* instructions for type consistency. Abstract interpretation goes instruction by instruction maintaining the actual state with the inferred types in the *Method* dataflow. Abstract interpretation takes into account how each instruction affects the types in the *Method* dataflow and correctly handles the convergence of different data flows in a given instruction.

This step includes, but is not limited to the following checks:

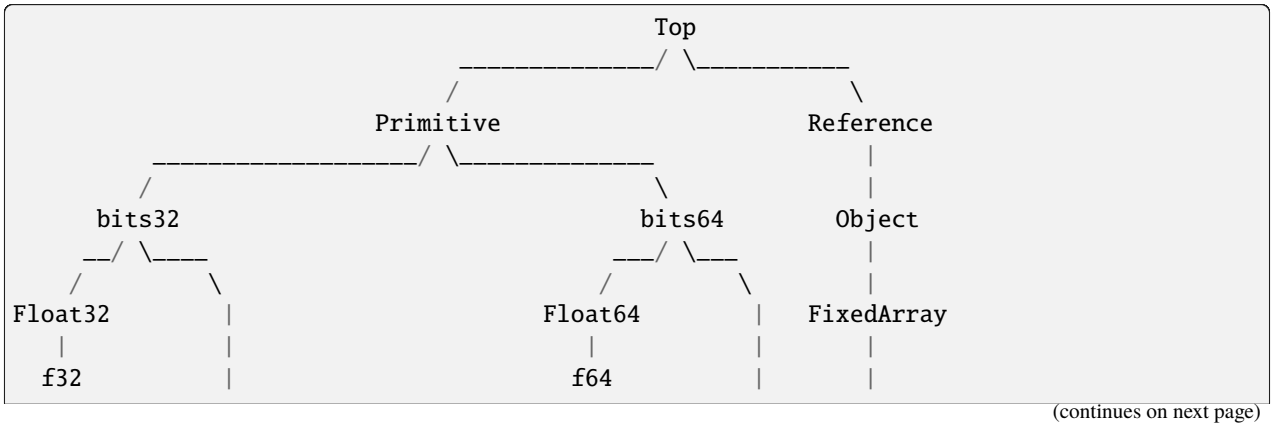
- Check that each accessed *Field* / *Class* / *Method* has corresponding access rights.
- Check that types expected as instruction input / method argument are correctly related via subtyping with actual types.
- Check that type of the value returned from the *Method* matches the it's signature.
- Check correctness of exception handlers.
- etc.

If any of the checks performed at this step fail, an error is thrown.

Verifier Type System

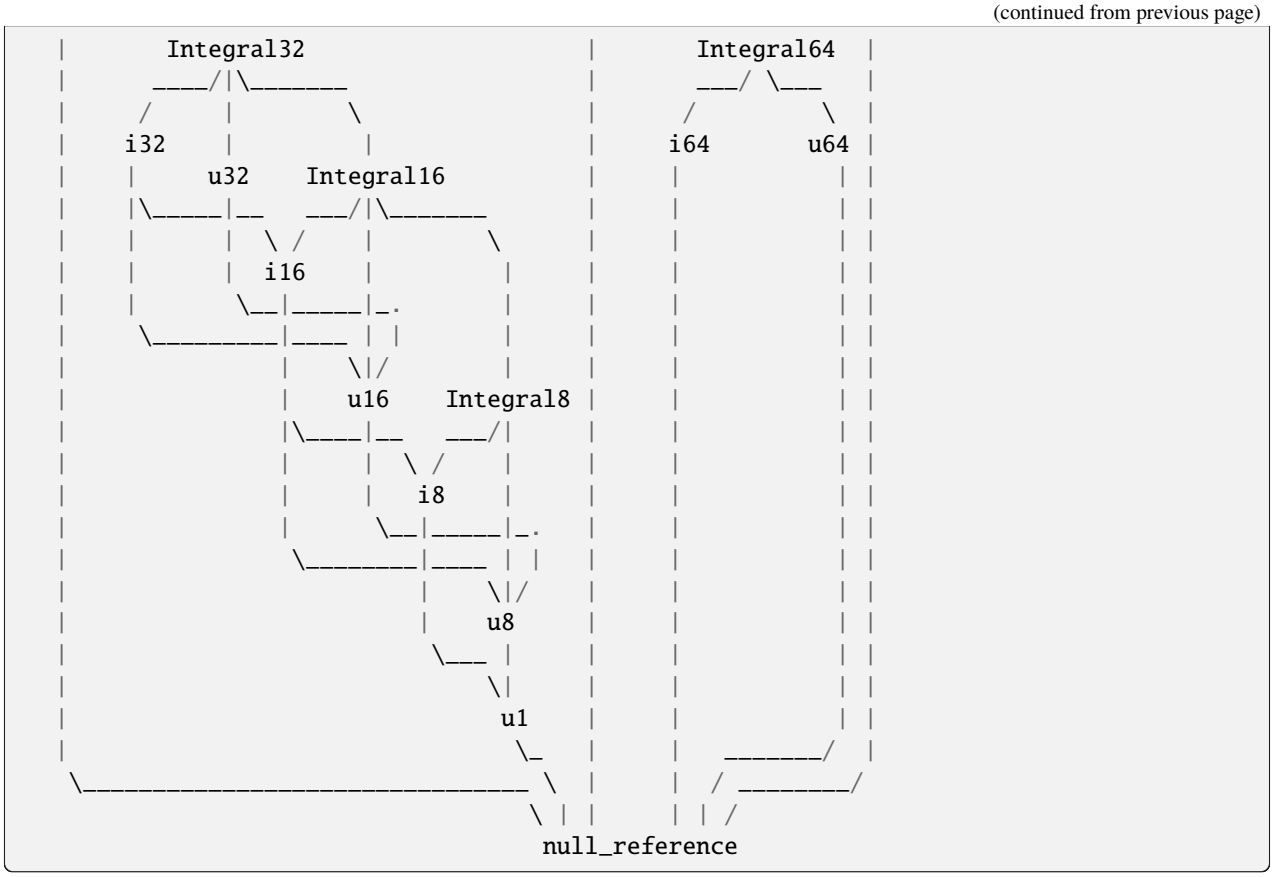
Below is listed the type hierarchy of the builtin types of the verifier.

Subtyping of the *user-defined types* is mandated by the *Runtime Type System*.



(continues on next page)

(continued from previous page)



CHAPTER

TWENTYTWO

GRAMMAR SUMMARY

```
literal: Literal;

identifier: Identifier;

type:
    annotationUsage?
    ( typeReference
    | 'readonly'? arrayType
    | 'readonly'? tupleType
    | functionType
    | functionTypeWithReceiver
    | unionType
    | keyofType
    | StringLiteral
    )
    | '(' type ')'
    ;

typeReference:
    qualifiedName typeArguments?
    ;

arrayType:
   type '[' ']'
   ;

functionType:
    '(' ftParameterList? ')' ftReturnType
    ;

ftParameterList:
    ftParameter (',' ftParameter)* (',' ftRestParameter)?
    | ftRestParameter
    ;

ftParameter:
    identifier ('?')? ':' type
    ;

ftRestParameter:
```

(continues on next page)

(continued from previous page)

```
    '...' ftParameter
    ;

ftReturnType:
    '=>' type
    ;

tupleType:
    '[' (type (',' type)* ','?)? ']'
    ;

unionType:
    type ('|' type)*
    ;

keyofType:
    'keyof' typeReference
    ;

qualifiedName:
  identifier ('.' identifier )*
  ;

typeDeclaration:
    classDeclaration
    | interfaceDeclaration
    | enumDeclaration
    | typeAlias
    ;

typeAlias:
    'type' identifier typeParameters? '=' type
    ;

variableDeclarations:
    'let' variableDeclarationList
    ;

variableDeclarationList:
    variableDeclaration (',' variableDeclaration)*
    ;

variableDeclaration:
    identifier ':' type initializer?
    | identifier initializer
    ;

initializer:
    '=' expression
    ;

constantDeclarations:
```

(continues on next page)

(continued from previous page)

```
    'const' constantDeclarationList
    ;

constantDeclarationList:
    constantDeclaration (',' constantDeclaration)*
    ;

constantDeclaration:
    identifier (':' type)? initializer
    ;

functionDeclaration:
    modifiers? 'function' identifier
    typeParameters? signature block?
    ;

modifiers:
    'native' | 'async'
    ;

signature:
    '(' parameterList? ')' returnType?
    ;

returnType:
    ':' (type | 'this')
    ;

parameterList:
    parameter (',' parameter)* (',' restParameter)? ','?
    | restParameter ','?
    ;

parameter:
    annotationUsage? (requiredParameter | optionalParameter)
    ;

requiredParameter:
    identifier ':' type
    ;

optionalParameter:
    identifier ':' type '=' expression
    | identifier '?' ':' type
    ;

restParameter:
    annotationUsage? '...' identifier ':' type
    ;

typeParameters:
    '<' typeParameterList '>'
```

(continues on next page)

(continued from previous page)

```
    ;

typeParameterList:
    typeParameter (',' typeParameter)*
    ;

typeParameter:
    ('in' | 'out')? identifier constraint? typeParameterDefault?
    ;

constraint:
    'extends' type
    ;

typeParameterDefault:
    '=' typeReference ('[]')?
    ;

typeArguments:
    '<' type (',' type)* '>'
    ;

explicitFunctionOverload:
    'overload' identifier overloadList
    ;

overloadList:
    '{' identifier (',' identifier)* ','? '}'
    ;


expression:
    primaryExpression
    | castExpression
    | instanceOfExpression
    | typeOfExpression
    | nullishCoalescingExpression
    | spreadExpression
    | unaryExpression
    | binaryExpression
    | assignmentExpression
    | ternaryConditionalExpression
    | stringInterpolation
    | lambdaExpression
    | lambdaExpressionWithReceiver
    | awaitExpression
    ;

primaryExpression:
    literal
    | namedReference
    | arrayLiteral
```

(continues on next page)

(continued from previous page)

```
    | objectLiteral
    | recordLiteral
    | thisExpression
    | parenthesizedExpression
    | methodCallExpression
    | fieldAccessExpression
    | indexingExpression
    | functionCallExpression
    | newExpression
    | ensureNotNullishExpression
    ;

binaryExpression:
    multiplicativeExpression
    | exponentiationExpression
    | additiveExpression
    | shiftExpression
    | relationalExpression
    | equalityExpression
    | bitwiseAndLogicalExpression
    | conditionalAndExpression
    | conditionalOrExpression
    ;

objectReference:
    typeReference
    | 'super'
    | primaryExpression
    ;

namedReference:
  qualifiedName typeArguments?
  ;

arrayLiteral:
    '[' expressionSequence? ']'
    ;

expressionSequence:
    expression (',' expression)* ','?
    ;

objectLiteral:
   '{' objectLiteralMembers? '}'
   ;

objectLiteralMembers:
   objectLiteralMember (',' objectLiteralMember)* ','?
   ;

objectLiteralMember:
   objectLiteralField | objectLiteralMethod
```

(continues on next page)

(continued from previous page)

```
    ;

objectLiteralField:
    identifier ':' expression
    ;

objectLiteralMethod:
    identifier typeParameters? signature block
    ;

spreadExpression:
    '...' expression
    ;

parenthesizedExpression:
    '(' expression ')'
    ;

thisExpression:
    'this'
    ;

fieldAccessExpression:
    objectReference ('.' | '?.') identifier
    ;

methodCallExpression:
    objectReference ('.' | '?.') identifier typeArguments? callArguments
    ;

functionCallExpression:
    expression ('?.' | typeArguments)? callArguments
    ;

callArguments:
    '(' argumentSequence? ')' trailingLambda?
    ;

argumentSequence:
    expression (',' expression)* ','?
    ;

trailingLambda:
    'async'? block
    ;

indexingExpression:
    expression ('?.')? '[' expression ']'
    ;

newExpression:
    newClassInstance
```

(continues on next page)

(continued from previous page)

```
    | newArrayInstance
    ;

newClassInstance:
    'new' typeArguments? typeReference arguments?
    ;

castExpression:
    expression 'as' type
    ;

instanceOfExpression:
    expression 'instanceof' type
    ;

typeOfExpression:
    'typeof' expression
    ;

ensureNotNullishExpression:
    expression '!'
    ;

nullishCoalescingExpression:
    expression '??' expression
    ;

unaryExpression:
    expression '++'
    | expression '--'
    | '++' expression
    | '--' expression
    | '+' expression
    | '-' expression
    | '~' expression
    | '!' expression
    ;

multiplicativeExpression:
    expression '*' expression
    | expression '/' expression
    | expression '%' expression
    ;

additiveExpression:
    expression '+' expression
    | expression '-' expression
    ;

shiftExpression:
    expression '<<' expression
    | expression '>>' expression
```

(continues on next page)

(continued from previous page)

```
    | expression '>>>' expression
    ;

relationalExpression:
    expression '<' expression
    | expression '>' expression
    | expression '<=' expression
    | expression '>=' expression
    ;

equalityExpression:
    expression ('==' | '===' | '!=' | '!==') expression
    ;

bitwiseAndLogicalExpression:
    expression '&' expression
    | expression '^' expression
    | expression '|' expression
    ;

conditionalAndExpression:
    expression '&&' expression
    ;

conditionalOrExpression:
    expression '||' expression
    ;

assignmentExpression
    : lhsExpression assignmentOperator rhsExpression
    | destructuringAssignment
    ;

assignmentOperator
    : '='
    | '+='  | '-='  | '*='   | '/='  | '%=' | `**=`
    | '<<=' | '>>=' | '>>>='
    | '&='  | '|='  | '^=' | `&&=` | `||=`
    | `??=`
    ;

lhsExpression:
    expression
    ;

rhsExpression:
    expression
    ;

ternaryConditionalExpression:
    expression '?' expression ':' expression
    ;
```

(continues on next page)

(continued from previous page)

```
stringInterpolation:
    '`' (BacktricksContentCharacter | embeddedExpression)* '`'
    ;

embeddedExpression:
    '${' expression '}'
    ;

lambdaExpression:
    annotationUsage? 'async'? lambdaSignature '=>' lambdaBody
    ;

lambdaBody:
    expression | block
    ;

lambdaSignature:
    '(' lambdaParameterList? ')' returnType?
    | identifier
    ;

lambdaParameterList:
    lambdaParameter (',' lambdaParameter)* (',' restParameter)? ','?
    | restParameter ','?
    ;

lambdaParameter:
    annotationUsage? (lambdaRequiredParameter | lambdaOptionalParameter)
    ;

lambdaRequiredParameter:
    identifier (':' type)?
    ;

lambdaOptionalParameter:
    identifier '?' (':' type)?
    ;

lambdaRestParameter:
    '...' lambdaRequiredParameter
    ;

constantExpression:
    expression
    ;

statement:
    expressionStatement
    | block
    | constantOrVariableDeclaration
    | ifStatement
```

(continues on next page)

(continued from previous page)

```
    | loopStatement
    | breakStatement
    | continueStatement
    | returnStatement
    | switchStatement
    | throwStatement
    | tryStatement
    ;

expressionStatement:
    expression
    ;

block:
    '{' statement* '}'
    ;

constantOrVariableDeclaration:
    annotationUsage?
    ( variableDeclaration
    | constantDeclaration
    )
    ;

ifStatement:
    'if' '(' expression ')' thenStatement
    ('else' elseStatement)?
    ;

thenStatement:
    statement
    ;

elseStatement:
    statement
    ;

loopStatement:
    (identifier ':')?
    whileStatement
    | doStatement
    | forStatement
    | forOfStatement
    ;

whileStatement:
    'while' '(' expression ')' statement
    ;

doStatement
    : 'do' statement 'while' '(' expression ')'
    ;
```

(continues on next page)

(continued from previous page)

```
forStatement:
    'for' '(' forInit? ';' expression? ';' forUpdate? ')' statement
    ;

forInit:
    expressionSequence
    | variableDeclarations
    ;

forUpdate:
    expressionSequence
    ;

forOfStatement:
    'for' '(' forVariable 'of' expression ')' statement
    ;

forVariable:
    identifier | ('let' | 'const') identifier (':' type)?
    ;

breakStatement:
    'break' identifier?
    ;

continueStatement:
    'continue' identifier?
    ;

returnStatement:
    'return' expression?
    ;

switchStatement:
    (identifier ':')? 'switch' '(' expression ')' switchBlock
    ;

switchBlock
    : '{' caseClause* defaultClause? caseClause* '}'
    ;

caseClause
    : 'case' expression ':' statement*
    ;

defaultClause
    : 'default' ':' statement*
    ;

throwStatement:
    'throw' expression
```

(continues on next page)

(continued from previous page)

```
    ;

tryStatement:
      'try' block catchClause? finallyClause?
      ;

catchClause:
      'catch' '(' identifier ')' block
      ;

finallyClause:
      'finally' block
      ;

classDeclaration:
    classModifier? ('class' | 'struct') identifier typeParameters?
      classExtendsClause? implementsClause?
      classMembers
    ;

classModifier:
    'abstract' | 'final'
    ;

classExtendsClause:
    'extends' typeReference
    ;

implementsClause:
    'implements' interfaceTypeList
    ;

interfaceTypeList:
    typeReference (',' typeReference)*
    ;

classMembers:
    '{'
       classMember* staticBlock? classMember*
    '}'
    ;

classMember:
    annotationUsage?
    accessModifier?
    ( constructorDeclaration
    | explicitConstructorOverload
    | classFieldDeclaration
    | classMethodDeclaration
    | explicitClassMethodOverload
    | classAccessorDeclaration
    )
```

(continues on next page)

(continued from previous page)

```
    ;


staticBlock:
    'static' block
      ;

accessModifier:
    'private'
    | 'protected'
    | 'public'
    ;

classFieldDeclaration:
    fieldModifier*
    identifier
    ( '?'? ':' type initializer?
    | '?'? initializer
    | '!' ':' type
    )
    ;

fieldModifier:
    'static' | 'readonly' | 'override'
    ;

classMethodDeclaration:
    methodModifier* typeParameters? identifier signature block?
    ;

methodModifier:
    'abstract'
    | 'static'
    | 'final'
    | 'override'
    | 'native'
    | 'async'
    ;

explicitClassMethodOverload:
    explicitClassMethodOverloadModifier*
    'overload' identifier overloadList
    ;

explicitClassMethodOverloadModifier: 'static' | 'async';


classAccessorDeclaration:
    classAccessorModifier*
    ( 'get' identifier '(' ')' returnType? block?
    | 'set' identifier '(' requiredParameter ')' block?
    )
```

(continues on next page)

(continued from previous page)

```
    ;

classAccessorModifier:
    'abstract'
    | 'static'
    | 'final'
    | 'override'
    ;

constructorDeclaration:
    'constructor' parameters constructorBody
    ;

constructorBody:
    '{' statement* '}'
    ;

explicitConstructorOverload:
    'overload' 'constructor' overloadList
    ;


interfaceDeclaration:
    'interface' identifier typeParameters?
    interfaceExtendsClause? '{' interfaceMember* '}'
    ;

interfaceExtendsClause:
    'extends' interfaceTypeList
    ;

interfaceMember
    : annotationUsage?
    ( interfaceProperty
    | interfaceMethodDeclaration
    | explicitInterfaceMethodOverload
    )
    ;

interfaceProperty:
    'readonly'? identifier '?'? ':' type
    | 'get' identifier '(' ')' returnType
    | 'set' identifier '(' requiredParameter ')'
    ;

interfaceMethodDeclaration:
    identifier signature
    | interfaceDefaultMethodDeclaration
    ;

explicitInterfaceMethodOverload:
    'overload' identifier overloadList'
```

(continues on next page)

(continued from previous page)

```
    ;

enumDeclaration:
    'const'? 'enum' identifier enumBaseType? '{' enumMemberList? '}'
    ;

enumBaseType:
    ':' type
    ;

enumMemberList:
    enumMember (',' enumMember)* ','?
    ;

enumMember:
    identifier initializer?
    ;

moduleDeclaration:
    moduleHeader? importDirective* (topDeclaration | topLevelStatements |␣
↪exportDirective)*
    ;

moduleHeader:
    'declare'? 'export'? 'module' moduleName
    ;

moduleName:
    StringLiteral
    ;

importDirective:
    'import' 'type'? bindings 'from' importPath
    ;

bindings:
    defaultBinding
    | (defaultBinding ',')? allBinding
    | (defaultBinding ',')? selectiveBindings
;

allBinding:
    '*' bindingAlias
    ;

bindingAlias:
    'as' identifier
    ;

defaultBinding:
    'type'? identifier
    ;
```

(continues on next page)

(continued from previous page)

```
selectiveBindings:
    nameBinding (',' nameBinding)*
    ;

nameBinding:
    'type'? identifier bindingAlias?
    | 'default' 'as' identifier
    ;

importPath:
    StringLiteral
    ;


topDeclaration:
    ('export' 'default'?)?
    annotationUsage?
    ( typeDeclaration
    | variableDeclarations
    | constantDeclarations
    | functionDeclaration
    | explicitFunctionOverload
    | namespaceDeclaration
    | ambientDeclaration
    | annotationDeclaration
    | accessorDeclaration
    | functionWithReceiverDeclaration
    )
    ;

namespaceDeclaration:
    'namespace' qualifiedName
    '{' (topDeclaration | topLevelStatements | staticBlock | exportDirective)* '}'
    ;

exportDirective:
    selectiveExportDirective
    | singleExportDirective
    | exportTypeDirective
    | reExportDirective
    ;

selectiveExportDirective:
    'export' selectiveBindings
    ;

singleExportDirective:
    'export'
    ( 'type'? identifier
    | 'default' (expression | identifier)
    | '{' identifier 'as' 'default' '}'
```

(continues on next page)

(continued from previous page)

```
    )
    ;

exportTypeDirective:
    'export' 'type' selectiveBindings
    ;

reExportDirective:
    'export'
    ('*' bindingAlias?
    | selectiveBindings
    | '{' 'default' bindingAlias? '}'
    )
    'from' importPath
    ;

topLevelStatements:
    statement*
    ;

ambientDeclaration:
    'declare'
    ( ambientConstantOrVariableDeclaration
    | ambientFunctionDeclaration
    | explicitFunctionOverload
    | ambientClassDeclaration
    | ambientInterfaceDeclaration
    | ambientEnumDeclaration
    | ambientNamespaceDeclaration
    | ambientAnnotationDeclaration
    | ambientAccessorDeclaration
    | typeAlias
    )
    ;

ambientConstantOrVariableDeclaration:
    'const'|'let' ambientConstantOrVariableList ';'
    ;

ambientConstantOrVariableList:
    ambientConstantOrVariable (',' ambientConstantOrVariable)*
    ;

ambientConstantOrVariable:
    identifier ':' type
    ;

ambientFunctionDeclaration:
    'function' identifier typeParameters? signature
    ;

ambientClassDeclaration:
```

(continues on next page)

(continued from previous page)

```
    'class'|'struct' identifier typeParameters?
    classExtendsClause? implementsClause?
    '{' ambientClassMember* '}'
    ;

ambientClassMember:
    ambientAccessModifier?
    ( ambientFieldDeclaration
    | ambientConstructorDeclaration
    | ambientMethodDeclaration
    | ambientClassAccessorDeclaration
    | ambientIndexerDeclaration
    | ambientCallSignatureDeclaration
    | ambientIterableDeclaration
    )
    ;

ambientAccessModifier:
    'public' | 'protected'
    ;

ambientFieldDeclaration:
    ambientFieldModifier* identifier ':' type
    ;

ambientFieldModifier:
    'static' | 'readonly'
    ;

ambientConstructorDeclaration:
    'constructor' parameters
    ;

ambientMethodDeclaration:
    ambientMethodModifier* identifier signature
    ;

ambientMethodModifier:
    'static'
    ;

ambientClassAccessorDeclaration:
    ambientMethodModifier*
    ( 'get' identifier '(' ')' returnType
    | 'set' identifier '(' requiredParameter ')'
    )
    ;

ambientIndexerDeclaration:
    'readonly'? '[' identifier ':' type ']' returnType
    ;
```

(continues on next page)

(continued from previous page)

```
ambientCallSignatureDeclaration:
    signature
    ;

ambientIterableDeclaration:
    '[Symbol.iterator]' '(' ')' returnType
    ;

ambientInterfaceDeclaration:
    'interface' identifier typeParameters?
    interfaceExtendsClause?
    '{' ambientInterfaceMember* '}'
    ;


ambientInterfaceMember
    : interfaceProperty
    | ambientInterfaceMethodDeclaration
    | ambientIndexerDeclaration
    | ambientIterableDeclaration
    ;

ambientInterfaceMethodDeclaration:
    'default'? identifier signature
    ;

ambientEnumDeclaration
    : 'enum' identifier enumBaseType? '{' ambientEnumMemberList? '}'
    | 'const' 'enum' identifier enumBaseType? '{' ambientConstEnumMemberList? '}'
    ;

ambientEnumMemberList:
    identifier (',' identifier)* ','?
    ;

ambientConstEnumMemberList:
    ambientConstEnumMember (',' ambientConstEnumMember)* ','?
    ;

ambientConstEnumMember:
    identifier '=' constExpression
    ;

ambientNamespaceDeclaration:
    'namespace' qualifiedName '{' ambientNamespaceElement* '}'
    ;

ambientNamespaceElement:
    ambientNamespaceElementDeclaration | selectiveExportDirective
;

ambientNamespaceElementDeclaration:
```

(continues on next page)

(continued from previous page)

```
    'export'?
    ( ambientConstantDeclaration
    | ambientFunctionDeclaration
    | ambientClassDeclaration
    | ambientInterfaceDeclaration
    | ambientNamespaceDeclaration
    | ambientAccessorDeclaration
    | 'const'? enumDeclaration
    | typeAlias
    )
    ;

ambientAccessorDeclaration:
    ( 'get' identifier '(' ')' returnType
    | 'set' identifier '(' requiredParameter ')'
    )
    ;

newArrayInstance:
    'new' arrayElementType dimensionExpression '(' arrayElement ')'
    ;

arrayElementType:
    typeReference
    | '(' type ')'
    ;

dimensionExpression:
    '[' expression ']'
    ;

arrayElement:
    expression
;

interfaceDefaultMethodDeclaration:
    'private'? identifier signature block
    ;

functionWithReceiverDeclaration:
    'function' identifier typeParameters? signatureWithReceiver block
    ;

signatureWithReceiver:
    '(' receiverParameter (', ' parameterList)? ')' returnType?
    ;

receiverParameter:
    annotationUsage? 'this' ':' type
    ;

functionTypeWithReceiver:
```

(continues on next page)

(continued from previous page)

```
    '(' receiverParameter (',' ftParameterList)? ')' ftReturnType
    ;

lambdaExpressionWithReceiver:
    annotationUsage?
    '(' receiverParameter (',' lambdaParameterList)? ')'
    returnType? '=>' lambdaBody
    ;

awaitExpression:
    'await' expression
    ;

annotationDeclaration:
    '@interface' identifier '{' annotationField* '}'
    ;

annotationField:
    identifier ':' type constInitializer?
    ;

constInitializer:
    '=' constantExpression
    ;

annotationUsage:
    '@' qualifiedName annotationValues?
    ;

annotationValues:
    '(' (objectLiteral | constantExpression)? ')'
    ;

ambientAnnotationDeclaration:
    'declare' annotationDeclaration
    ;

Identifier:
  IdentifierStart IdentifierPart*
  ;

IdentifierStart:
  UnicodeIDStart
  | '$'
  | '_'
  | '\\' EscapeSequence
  ;

IdentifierPart:
  UnicodeIDContinue
  | '$'
  | ZWNJ
```

(continues on next page)

(continued from previous page)

```
  | ZWJ
  | '\\' EscapeSequence
  ;

ZWJ:
 '\u200C'
;

ZWNJ:
 '\u200D'
;

UnicodeIDStart
  : Letter
  | ['$']
  | '\\' UnicodeEscapeSequence;

UnicodeIDContinue
  : UnicodeIDStart
  | UnicodeDigit
  | '\u200C'
  | '\u200D';

UnicodeEscapeSequence:
  'u' HexDigit HexDigit HexDigit HexDigit
  | 'u' '{' HexDigit HexDigit+ '}'
  ;

Letter
  : UNICODE_CLASS_LU
  | UNICODE_CLASS_LL
  | UNICODE_CLASS_LT
  | UNICODE_CLASS_LM
  | UNICODE_CLASS_LO
  ;

UnicodeDigit
  : UNICODE_CLASS_ND
  ;


Literal:
  IntegerLiteral
  | FloatLiteral
  | BigIntLiteral
  | BooleanLiteral
  | StringLiteral
  | MultilineStringLiteral
  | NullLiteral
  | UndefinedLiteral
  | CharLiteral
  ;
```

(continues on next page)

(continued from previous page)

```
IntegerLiteral:
  DecimalIntegerLiteral
  | HexIntegerLiteral
  | OctalIntegerLiteral
  | BinaryIntegerLiteral
  ;

DecimalIntegerLiteral:
  '0'
  | DecimalDigitNotZero ('_'? DecimalDigit)*
  ;

DecimalDigit:
  [0-9]
  ;

DecimalDigitNotZero:
  [1-9]
  ;

HexIntegerLiteral:
  '0' [xX]  ( HexDigit
  | HexDigit (HexDigit | '_')* HexDigit
  )
  ;

HexDigit:
  [0-9a-fA-F]
  ;

OctalIntegerLiteral:
  '0' [oO] ( OctalDigit
  | OctalDigit (OctalDigit | '_')* OctalDigit )
  ;

OctalDigit:
  [0-7]
  ;

BinaryIntegerLiteral:
  '0' [bB] ( BinaryDigit
  | BinaryDigit (BinaryDigit | '_')* BinaryDigit )
  ;

BinaryDigit:
  [0-1]
  ;

FloatLiteral:
    DecimalIntegerLiteral '.' FractionalPart? ExponentPart? FloatTypeSuffix?
    | '.' FractionalPart ExponentPart? FloatTypeSuffix?
```

(continues on next page)

(continued from previous page)

```
    | DecimalIntegerLiteral ExponentPart? FloatTypeSuffix
    ;

ExponentPart:
    [eE] [+-]? DecimalIntegerLiteral
    ;

FractionalPart:
    DecimalDigit
    | DecimalDigit (DecimalDigit | '_')* DecimalDigit
    ;

FloatTypeSuffix:
    'f'
    ;

BigIntLiteral: DecimalIntegerLiteral 'n';

.. BigIntLiteral: IntegerLiteral 'n';

BooleanLiteral:
    'true' | 'false'
    ;

StringLiteral:
    '"' DoubleQuoteCharacter* '"'
    | '\'' SingleQuoteCharacter* '\''
    ;

DoubleQuoteCharacter:
    ~["\\\r\n]
    | '\\' EscapeSequence
    ;

SingleQuoteCharacter:
    ~['\\\r\n]
    | '\\' EscapeSequence
    ;

EscapeSequence:
    ['"bfnrtv0\\]
    | 'x' HexDigit HexDigit
    | 'u' HexDigit HexDigit HexDigit HexDigit
    | 'u' '{' HexDigit+ '}'
    | ~[1-9xu\r\n]
    ;

MultilineStringLiteral:
    '`' (BacktricksContentCharacter)* '`'
    ;

BacktricksContentCharacter:
```

(continues on next page)

(continued from previous page)

```
    ~[`\\\r\n]
    | '\\' EscapeSequence
    | LineContinuation
    ;

 LineContinuation:
    '\\' [\r\n\u2028\u2029]+
    ;

NullLiteral:
    'null'
    ;

UndefinedLiteral:
    'undefined'
    ;

CharLiteral:
    'c\'' SingleQuoteCharacter '\''
    ;
```

CHAPTER

TWENTYTHREE

BINARY COMPATIBILITY

The ArkTS allows source code of a single runnable program to be compiled piece by piece. Such property allows source code authors to develop and distribute code independently. However, certain limitations have to be met for the existing programs to remain runnable without the recompilation of all the source code that constitutes them.

Binary compatibility is different from the *source compatibility*, which allows the dependent source code to remain compilable without any *compile-time errors*. Certain *source compatible* changes may be not *binary compatible* and vice versa.

Note

Certain *binary incompatible* changes may violate the type system invariants of the execution environment, making the execution of binary code impossible. In that case, the runtime verification system (see *Verification*) is responsible for the detection and rejection of such binary code.

This chapter defines the *binary compatible* change of the source code as a change that allows all the dependent programs with all the *types* and *namespaces loadable* and *resolvable* to preserve this property.

Type or *namespace* is *loadable* if the code that depends on such entity and none of its members can be executed within the running program, while the *Static Initialization* of the entity *completes*. If the entity is not *loadable*, the *Static Initialization* does not *complete* and throws an error.

Type or *namespace* is *resolvable* if all of its members, such as *namespace* variables and functions or *class* fields and *methods*, can be accessed with no *resolution* errors, i.e., errors that do not appear within the consistently compiled programs. *Resolvable* type is always *loadable*.

```
// source1.ets
export interface I {
  m(): void
}

// source2.ets
export namespace N {

  // The type I must be loadable, otherwise the f and the whole N is not loadable.
  //
  // Example case: the definition of I is removed during the change in
  // the source1.ets, while the code of function f was not recompiled.
  export function f(i: I) {

    // I.m must be resolvable for the actual type of I, otherwise an attempt
    // to call i.m will throw a runtime error.
```

(continues on next page)

(continued from previous page)

```
    //
    // Example case: the class that was compiled with an older source code
    // of source1.ets, where I does not declare the method m, does not provide
    // an implementation of the method I.m.
    return i.m()
  }
}

// app.ets

class A implements I {
  m() { }
}

try {
  N.f(new A())
} catch (e) {
  // Some platform-specific errors may appear if the binary compatibility
  // was broken by changes in source1.ets or source2.ets
  //
  // While the recovery for the running program may be possible in some
  // circumstances, this is a detail of the platform whether such
  // program can run or not, and whether it will be terminated during
  // the execution or not.
}
```

Source code changes may produce specific errors during the execution that may not occur if the whole executed program was compiled without any intermediate source code changes. The resulting errors and conditions may cause some entity to remain *resolvable*, not *resolvable* but *loadable*, or not *loadable*. For example,

- If a *public* or *exported* type is removed from the source code, the code that accesses the type becomes invalid, making some dependent code not *loadable*.
- *Dynamic Method Resolution* may fail if the method definition is missing or some method definitions are conflicting. The fail results in an error thrown during an invocation of the method, while the considered class remains *loadable*, yet not *resolvable*.
- Changes in the type hierarchy may cause a *type circularity*. Types which constitute the *circularity* are not *loadable*.

23.1 Classification of Source Code Changes

For bodies of *Method Declarations* and *Function Declarations*, *Top-level Statements*, initialization expressions of *Top-level Variables* and *static class fields*:

- If the *Effective Type* of the one *Method* or *Function* declaration, *field* or *variable* is not affected, the change does not affect *binary compatibility*.
- Otherwise, see other sections for the changes in the *Effective Type*.

For *Namespace Definitions*:

Changes that don't affect the binary compatibility:

- Renaming or removing of a *namespace* member which is not *exported*.
- Adding a new member *namespace*.

- Adding a new *function* or *variable*.
- Adding a new member type definition, such as *class*, *interface*, *enum* or *type alias*.

Some of the changes that break the binary compatibility:

- Changing the name or removing of an *exported namespace*.
- Changing the *Effective Type* of an *exported function* or *variable*.
- Changing the name or removing of an *exported function* or *variable*.
- Changing the name or removing of an *exported class*, *interface*, *enum*, *type alias* or *namespace*.

For *Class* and *Interface Declarations*:

Changes that don't affect the binary compatibility:

- Reordering the types in *extends* or *implements* clause.
- Inserting a previously nonexistent interface or class with no members to the type hierarchy.
- Reordering of members, such as *fields*, *methods*.
- Adding, removing or renaming a *private* member.
- Adding a new *field* to the *class*.

Some of the changes that can make some compiled code not *resolvable*:

- If some derived class contains several method definitions of the same name, the following conditions may cause a conflict during the method resolution:
  - Adding a new *public* method to an *exported* class.
  - Moving a method upwards in the class hierarchy.
  - Adding a new default method or an optional property to the interface may cause a conflict during the method resolution.
- Adding a *required property* or a *method* to an interface without supplying a *default implementation* for it.
- Adding an *abstract* method to an existing *exported abstract* class.

Some of the changes that can make some compiled code not *loadable*:

- Changing *Effective Type* of a *public* or *protected method* or *field*.
- Changing the name or removing a *public* or *protected method* or *field*.
- Adding a *final* modifier to an existing *exported* class.

For *enumeration declarations*:

Changes that do not affect binary compatibility:

- Adding new enum constants.
- Changing the value of enum constants while preserving its type.

CHAPTER

TWENTYFOUR

BUILD SYSTEM

The Build System of the ArkTS language defines the following:

- *Package Definition* that introduces the concept of a package as a distributable unit composed of one or more modules.
- *Module Visibility* that determines whether a module is accessible from other packages based on the presence of the modifier `export` in the module header (e.g.: `export module "name"`). Modules with no modifier `export` are internal to the package.
- *Runtime Name Formation* that defines the fully qualified name (see *Fully Qualified Name*) for each entity. This name is used for runtime identification and access control.
- *ImportPath Resolution Rules* that defines how an *importPath* (see *Import Path*) is resolved to an actual module.

24.1 Package Definition

Package is defined as an entity that groups together one or more modules. *Package* is a unit of distribution. Each module (see *Module Declarations*) belongs to exactly one package.

A package is described by a configuration file that provides the build system with all information necessary to compile, link, and distribute the package. The file specifies the following:

- *Package name*, i.e., a string that uniquely identifies the package in the build. The name is used as a prefix when forming runtime names (see *Runtime Name Formation*).
- *Source files*, i.e., the set of the ArkTS files that constitute the package. The build system compiles these files and combines them into the binary file (see *Binary File Format*).
- *Dependencies*, i.e., a set of other packages on which the package depends. The build system uses this information to locate those packages during compilation, and to ensure that their exported declarations are available for *importPath* (see *Import Path*).
- *ImportPath resolution rules*, i.e., the rules affecting how *importPath* (see *Import Path*) is resolved to modules within the package or in external packages. The rules are applied by the build system during compilation to locate imported modules.

24.2 Module Visibility

Module visibility determines whether a module can be accessed from modules residing in other packages. The visibility of a module is determined by the modifier `export` in the module header (see *Module Header*). If the module has no module header (see *Module Header*), then the visibility of the module is determined similarly to that of a module without the modifier `export`.

A module can be *exported* or *internal*:

- *Exported* is a module declared with the modifier `export`. An exported module is accessible from modules of other packages.
- *Internal* is a module declared without the modifier `export`. An internal module is accessible only from other modules belonging to the same package.

Attempting to import an *internal* module from another package causes a compile-time error.

An *exported* module with a *module header* is represented in the example below:

```
export module "x"
// module contents
```

An *internal* module with a *module header* is represented in the example below:

```
module "y"
// module contents
```

An *internal* module with no *module header* is represented in the example below:

```
// module contents
```

A *public API of a package* in the build system is any declaration exported from an exported module. If a signature of an exported function, method, class, or interface specifically includes a unexported type (e.g., a type defined in an internal module), then a compile-time error occurs. This rule is consistent with the general requirement for exported declarations as discussed in detail in *Exported Declarations*. The build system verifies this property at compile time to ensure that the public interface of the package remains self-contained, and that runtime visibility checks (see *Verification*) never fail due to leaks.

Internal type protection from package leaks is represented in the example below:

```
// file1.ets
module "y"          // Internal module "y" (not exported)
export class InternalClass1 {}

// file2.ets        // Internal module (not exported)
export class InternalClass2 {}

// file3.ets
export module "x"   // Exported module "x"
import { InternalClass1 } from "./file1"        // OK, import is allowed
import { InternalClass2 } from "./file2"        // OK, import is allowed

export function foo(x: InternalClass1): void {} // Compile-time error, exported␣
↪function signature contains reference to an unexported type
export function foo(x: InternalClass2): void {} // Compile-time error, exported␣
↪function signature contains reference to an unexported type
```

24.3 Runtime Name Formation

The build system is responsible for constructing the runtime name (see *Runtime Name*) of each entity.

Forming a runtime name (see *Runtime Name*) from the package name, the module name (see *Module Header*), and the qualified name of an entity is defined by the following rules:

```
if M is not set:
    M = F

if M is an empty string, and Q is not set:
    R = P
if M is an empty string:
    R = P + ":" + Q
else:
    R = P + ":" + M + "." + Q

R = R.replace("/", ".")
```

– where:

- `M` is the module name declared in the module header (see *Module Header*).
- `F` is the relative path to the file in the package (see *Package Definition*) with its extension deleted.
- `R` is the runtime name (see *Runtime Name*) of an entity.
- `P` is the name of the package that contains an entity (see *Package Definition*).
- `Q` is the qualified name of an entity within the module. The name consists of dot-separated identifiers (see *Names*) except functions, variables, methods, and static methods.

If the resulting `R` coincides with the runtime name (see *Runtime Name*) of any other entity in the same package, then a compile-time error occurs.

Forming a runtime name is represented in the example below:

```
// packageName: pkg1

// src/file1.ets
export module ""            // Runtime Name: "pkg1"

export class A {}           // Runtime Name: "pkg1:A"
export namespace ns1 {      // Runtime Name: "pkg1:ns1"
    export namespace ns2 {  // Runtime Name: "pkg1:ns1.ns2"
        exprot class B {}   // Runtime Name: "pkg1:ns1.ns2.B"
    }
    export class C {}       // Runtime Name: "pkg1:ns1.C"
}

// src/file2.ets
export module "x"           // Runtime Name: "pkg1:x"

export class A {}           // Runtime Name: "pkg1:x.A"
```

(continues on next page)

(continued from previous page)

```
export namespace ns1 {       // Runtime Name: "pkg1:x.ns1"
    export namespace ns2 {   // Runtime Name: "pkg1:x.ns1.ns2"
        exprot class B {}    // Runtime Name: "pkg1:x.ns1.ns2.B"
    }
    export class C {}        // Runtime Name: "pkg1:x.ns1.C"
}

// src/file3.ets
export class A {}            // Runtime Name: "pkg1:src.file3.A"
export namespace ns1 {       // Runtime Name: "pkg1:src.file3.ns1"
    export namespace ns2 {   // Runtime Name: "pkg1:src.file3.ns1.ns2"
        exprot class B {}    // Runtime Name: "pkg1:src.file3.ns1.ns2.B"
    }
    export class C {}        // Runtime Name: "pkg1:src.file3.ns1.C"
}
```

24.4 ImportPath Resolution Rules

The build system resolves importPath (see *Import Path*) according to the rules below.

Step 1. The build system tries to resolve the *import path* inside its package by using one of the following:

- *File-relative path*. If the *import path* (see *Import Path*) starts with the prefixes `./` or `../`, then the *import path* points at the file path that resolves the relative path to the location of the importing file inside the package. If no file exists, then a compile-time error occurs.
- *Module name*. If the *import path* starts with the prefix `//`, then the *import path* refers to the *module name* declared in the *module header* (see *Module Header*) and located within the current package. If no module exists, then a compile-time error occurs.
- *Full module name*. If the *import path* equals `"package_name/module_name"`, where `package_name` is the name of the current package, and `module_name` is the name of one of the modules of the current package declared in the *module header* (see *Module Header*), then the *import path* refers to the corresponding module.

Step 2. The build system tries to resolve the *import path* to the external module by using one of the following:

- *Aliases section*. If the *import path* equals `"alias_name/module_name"`, where `alias_name` is an alias of the name of an external package, and `module_name` is the name one of the modules of that package, then the *import path* refers to the corresponding module.
- *Dependencies section*. If the *import path* equals `"package_name/module_name"`, where `package_name` is the name of an external package, and `module_name` is the name of one of the modules of that package, then the *import path* refers to the corresponding module.

If the *import path* fails to resolve, then a compile-time error occurs.

Resolving an import within a single package is represented in the example below:

```
// packageName: pkg1

// src/file1.ets
module "x"
```

(continues on next page)

(continued from previous page)

```
export function foo() {}

// src/file2.ets
export function foo() {}

// src/file3.ets
import {...} from "./file1"       // OK, importPath is resolved to "file1.ets" by the relative path
import {...} from "./file2"       // OK, importPath is resolved to "file2.ets" by the relative path
import {...} from "../src/file1"  // OK, importPath is resolved to "file1.ets" by the relative path
import {...} from "../src/file2"  // OK, importPath is resolved to "file2.ets" by the relative path
import {...} from "//x"           // OK, importPath is resolved to "x" by the module name
import {...} from "//file2"       // Compile-time error, importPath refers to a non-existent module inside the package
import {...} from "pkg1/x"        // OK, importPath is resolved to "x" by the full module name
import {...} from "pkg1/file2"    // Compile-time error, importPath refers to a non-existent module inside the package
```

CHAPTER

TWENTYFIVE

CONTRIBUTORS

Language design lead:

- Nedoria Aleksei

Contributors:

- Bronnikov Georgy
- Cherkashin Vyacheslav
- Gavrin Evgeny
- Grebenschikov Konstantin
- Fang Lou
- Huo Qingyi
- Kanatov Alexey
- Nedoria Aleksei
- Olshevsky Vladimir
- Pavlyuk Alexander
- Pei Jiajun
- Polyakov Alexander
- Pukhov Vsevolod
- Qiu Yu
- Rubanov Vladimir
- Soldatov Anton
- Solomennikov Dmitry
- Trubenkov Dmitrii
- Velikanov Michael
- Wan Mingxiang
- Wu Hongfei
- Xian Yuqiang
- Zouev Evgeniy

Technical writer:

- Baranov Dmitry

INDEX

A

abrupt completion, 97, 99, 100, 103, 114, 120, 126, 128, 140–142, 165–168, 173, 178, 190, 191, 345
abstract, 216
abstract class, 108, 194, 195, 217, 386
abstract concept, 2
abstract data structure, 2
abstract declaration, **3**, 233
abstract function, 327, 346
abstract keyword, 363
abstract method, 108, 123, 195, 216–218, 227
abstract method call, 123
abstract modifier, 194, 195, 216, 217, 219
abstract notion, 2
abstract symbol, 3
abstraction, 2
access, 36, 39–41, 49, 55, 56, 62, 63, 75, 95, 120, 121, 125, 126, 129, 165, 168, 189, 205–211, 222, 227, 230, 234, 236, 245, 248, 249, 251, 252, 256, 257, 263, 266, 268, 279, 280, 300, 321, 322, 342–344, 347, 365, 366, 377, 381, 385, 390, 391
access constructor, 94
access control, 463
access expression, 166
access modifier, 62, 193, 206–208, 219, 222, 227, 234, 300, 357
accessibility, 50, 56, 59, 62, 63, 75, 109, 120, 172, 184, 195, 197, 205–209, 222, 227, 230, 234, 248, 249, 252, 256, 257, 262, 263, 266, 268, 280, 300, 344, 347, 353, 365, 366, 377
accessible constructor, 207, 227
accessible declaration, 249, 257, 280
accessible entity, 63
accessible interface type, 197
accessible member, 227
accessible member field, 109, 120
accessible method, 357
accessible scope, 62
accessible type, 208
accessor, 47, 110, 113, 121, 193, 203–206, 214, 220, 221, 227, 231–233, 300, 364, 373, 374
accessor declaration, 110, 121, 206, 214, 257, 373, 374
accessor modifier, 219, 220
accessor with receiver, 257
addition, 13, 153
additive expression, 30, 87, 152, 153, 174
additive operator, 29, 31, 152, 153, 174
alias, 23, 24, 27, 40, 43, 47, 64, 65, 195, 230, 256, 358
alignment, 161, 323
allocation, 173
alpha-numeric character, 254
ambient, 271
ambient accessor declaration, 274
ambient annotation, 382
ambient call signature, 276
ambient call signature declaration, 276
ambient class, 275, 277, 278
ambient class declaration, 274–276
ambient constant, 272
ambient constructor, 271
ambient constructor declaration, 274
ambient context, 273, 275–277, 280
ambient declaration, 257, 262, 271, 383
ambient field declaration, 274
ambient function, 271
ambient function declaration, 273
ambient indexer, 275, 277
ambient indexer declaration, 275
ambient interface, 277
ambient interface declaration, 277, 278
ambient iterable, 277
ambient iterable declaration, 277, 279
ambient method, 271, 275
ambient method declaration, 274
ambient namespace, 249, 279, 280
ambient namespace declaration, 279, 280
ambient overload function, 273
ambient overload function declaration, 273
ambient variable, 272
AND operator, 162
annotation, 26, 35, 66, 68, 171, 180, 184, 205, 218, 231,

257, 282, 297, 327, 352, 370, 377–386
annotation declaration, 377–379, 381, 383
annotation field, 378, 379
annotation name, 381
anonymous class, 111, 112, 114
anonymous type, 27, 64
any, 24
Any type, 47
any type, 78
API, 329
arbitrary large integer, 37
arbitrary signature, 349, 350
argument, 44, 70–74, 100, 115, 118, 123, 124, 130, 227, 268, 283, 349, 359, 364, 371, 372, 377, 390
argument expression, 87
argument type, 80, 206, 296, 391
arithmetic operator, 87
ArkUI code, 390
ArkUI plugin, 390
ArkUI type, 390
array, 24, 27, 33, 39, 40, 64, 67, 71, 77, 78, 81, 82, 97, 100, 103, 104, 118, 126, 130, 165, 167, 183, 284, 293, 296, 339, 344, 345, 378, 390
array access, 126
array access expression, 97
array argument, 73
array bounds checking, 241
array creation, 339, 344, 345
array creation expression, 130, 344, 345
array declaration, 39, 342
array dimension, 344
array element, 33, 39–41, 65, 85, 97, 103, 104, 107, 125, 165, 167, 168, 293, 342, 344
array element type, 97, 293
array indexing, 125
array indexing expression, 97, 126, 165, 167
array initialization, 104
array initializer, 103
array instance, 38, 130, 342–344
array length, **3**, 38–40, 97, 103, 126, 165, 167, 342, 343
array literal, 47, 49, 85, 97, 103, 104, 106, 107, 115, 133, 281, 297, 339, 342, 344, 380, 381
array literal expression, 103
array operation, 40
array reference expression, 97, 126, 165
array reference subexpression, 167
array size, 339
array type, **3**, 23, 25, 26, 33, 38–40, 71–73, 81, 103, 115, 125, 126, 284, 294, 296, 339, 342, 368, 380, 381, 385
array value, 38
assign, 282, 342
assignability, 39–41, 43, 53, 54, 66, 68, 72, 73, 76, 78, 82, 83, 86, 109, 133, 165, 187, 189, 191, 229, 237, 282, 284, 294, 296, 342, 344, 347, 352
assignable type, 53, 54, 76, 82, 296
assignment, 34, 40, 41, 44, 49, 57, 66, 71, 85, 88, 95–97, 100, 104, 115, 129, 164–168, 184, 189, 210, 211, 282, 294, 344, 374
assignment context, 85
assignment expression, 67, 164–168
assignment operator, 66, 99, 164, 165, 167
assignment-like context, 85, 88, 282–284
assignment-like contexts, 68
associativity, 95, 99, 147, 153, 162, 164
asymmetric relationship, 294
async function, 51, 327–329, 346
async lambda, 327
async mark, 171
async method, 217, 327, 347
async modifier, 217, 273, 327
async type, 299
asynchronous API, 329
asynchronous execution, **3**
asynchronous operation, 329
asynchronous programming, 326
automatic transition, 2
await expression, 328
await operator, 95
awaited, 51

B

backslash, 19, 20
backslash character, 254
backspace, 20
backtick, 20, 170
backward compatibility, 327
balanced braces, 179
base, 151, 295
base class, 195, 213, 226, 241, 295, 303
base type, 90, 294, 304
base URL, 255
bigint, 24, 28, 29, 154
bigint comparison, 156
bigint literal, 18, 37
bigint operand, 156
bigint type, 33, 37, 144, 154, 156, 158, 162
binary, 14
binary expression, 138
binary numeric expression, 283
binary operation, 167, 168
binary operator, 31, 86, 99, 147–149, 151–153, 283
bind all, 63
binding, 249–256, 264
bitwise AND operand, 162
bitwise complement, 144
bitwise complement expression, 144, 147

bitwise complement operator, 29, 31, 144
bitwise exclusive OR operand, 162
bitwise expression, 87, 162, 163
bitwise inclusive OR operand, 162
bitwise logical AND operator, 154
bitwise operator, 95, 162, 174
block, 63, 69, 121, 123, 172, 179, 181, 187, 189, 190, 218, 219, 363, 364, 372
block notation, 371
block of code, 222
block scope, 63, 180, 181
block statement, 179
body, 63, 69, 216, 233
Boolean, 32
boolean, 29, 30, 32, 57, 86, 131, 290
boolean comparison, 157
Boolean literal, 18
boolean logic, 323
boolean logical expression, 163
boolean operand, 163, 164
boolean operator, 163
boolean relational operator, 157
boolean type, 28, 31, 75, 144, 155, 158, 162–164, 169, 180, 182, 323, 378
boolean value, 169
bound, 289
bound entity, 252, 255
bounded instance, 161
bounded object, 161
brace, 103
bracket, 103, 125
break, 188
break statement, 181, 184, 185, 187, 188
bridge method, 300, 391
build system, 463
built-in annotation, 383
built-in array, 27
built-in getter, 53
built-in setter, 53
built-in type, 37, 39, 50, 342, 343
byte, 28, 29, 57, 89, 91
bytecode, 384, 385
bytecode file, 384

C

C_JOB, **5**
C_WORKER, **8**
call, 73, 95, 115, 121–124, 214, 226, 227, 245, 252, 266, 307, 315, 337, 348, 349, 351, 353, 360, 365, 372, 383, 390
call argument, 124, 296
call context, 85
call expression, 172, 348
call method, 210
call parameter type, 104
call site, 118, 353, 361
callable class type, 348
callable type, 276, 348–351, 358
callee, 124
caller scope, 191
captured by lambda, 172, 173
captured variable, 173
carriage return character, 10
case sensitivity, 254
cast, 30, 31, 133
cast conversion, 134
cast expression, 50, 97, 104, 133, 134, 174, 319, 321
cast operator, 29, 31, 95, 133
casting conversion, **4**, 91, 174
catch clause, 189–191
catch identifier, 189
chaining, 95
chaining operator, 94, 121, 123, 125, 126, 129, 168
char, 14, 57
char conversion, 88
char literal, 340
char type, 340
character, 3, 9, 20, 36, 340
character type, 28
check, 134, 211, 296
circular dependency, 210
circular reference, 45
class, 27, 33, 41, 49, 50, 53, 59, 62, 77–82, 94, 101–103, 107–109, 114, 119–123, 126, 130, 131, 160, 187, 194–197, 203–205, 207–210, 212, 215–218, 220, 226, 227, 229, 230, 232, 241, 257, 289, 300, 303, 304, 308, 321, 322, 344, 345, 347, 348, 358, 360, 361, 363, 364, 367, 371, 380, 385
class accessor, 214, 219
class accessor declaration, 204, 219
class body, 193, 205, 207, 209, 224, 227
class constructor, 193, 210
class declaration, 27, 64, 193, 194, 197, 205, 208, 211, 215, 227, 363, 377, 380, 390
class declaration body, 215
class declaration scope, 206
class extension, 195, 286, 363
class extension clause, 195
class field, 109, 206, 385
class fields, 209
class getter, 385
class implementation clause, 197
class instance, 62, 100, 107, 114, 173, 205, 206, 208–210, 221, 222, 275, 277, 309
class instance creation expression, 130, 131
class instance expression, 130
class instance method, 76

class instantiation, 222
class keyword, 390
class level scope, **4**, 62, 63
class member, 62, 63, 193, 204–208, 356, 385
class method, 204, 214, 220, 275, 327, 352, 356, 385
class method declaration, 214, 215
class method overload declaration, 361
class name, 194, 230
class Object, 227
class setter, 385
class type, 23, 25, 33, 45, 47, 52, 53, 83, 108, 119, 121, 133, 194, 195, 197, 304, 345, 348, 350, 363, 368
class-composite context, 109
class-level scope, 193
clause, 195
closure, 284
code readability, 297
comma, 13
comma-separated argument expression, 100
comma-separated list, 108
command-line argument, 269
comment, 2, **4**, 9, 10, 22
common subset, 1
commutative operation, 147, 153
commutative operator, 158, 162
comparison, 13, 32, 155, 158, 160
comparison operator, 29, 31
compatibility, 24, 28, 37, 39, 41, 50, 55, 78, 96, 106, 124, 234, 235, 246, 254, 271, 275–277, 279, 296, 306, 322, 327, 342, 348, 352, 368
compatible code, 348
compatible expression, 96
compilation, 125, 227, 254, 390
compilation environment, 255
compile time, 23, 96, 121, 135, 174, 210, 309, 344, 352, 360, 367, 378, 384
compile-time error, **4**
compile-time warning, **4**
compile-time, 122, 310, 340
compile-time error, 16–18, 36, 37, 39–41, 43, 45, 47, 48, 50, 52–55, 60, 61, 65–72, 75, 79–83, 86, 89, 94, 96, 97, 101, 102, 104–106, 108–112, 114, 118–125, 127, 129–131, 133, 135, 138–142, 144, 146, 148–155, 157, 160, 162–165, 168, 171, 172, 174, 180–189, 194, 195, 197–204, 208, 209, 211–220, 222, 223, 225–227, 230, 231, 234–237, 244–248, 250, 251, 254, 257, 258, 263, 264, 266–268, 271, 272, 278, 279, 281–283, 295, 296, 298–304, 307–310, 316, 327, 328, 340, 343, 344, 346–360, 362–364, 368, 369, 372–375, 377–380, 382–385, 390, 464–466
compile-time feature, 2
compile-time polymorphism, 307
compile-time warning, 37, 129, 132–134, 137, 138, 159, 210, 223, 250, 315, 316
compiler, 23, 65, 115, 127, 128, 210, 211, 224, 254, 268, 299, 346, 347, 383, 384, 389–391
compiler check, 384
compiler environment, 255
compiler-known signature, 346, 347
complement expression, 144
complement operator, 32
completion, 140, 241
completion failure, 241
compliance, 91
composite literal context, 85
compound assignment expression, 167
compound assignment operator, 167, 168
compound-assignment operator, 99
concatenation, 31, 32, 85, 170
concatenation operator, 36
concrete method, 216
concurrency, 326
concurrent execution, **4**, 322
conditional evaluation, 163
conditional expression, 144, 180
conditional operator, 29, 31, 99, 174
conditional-and expression, 87, 163, 323, 324
conditional-and operator, 32, 138, 163, 174
conditional-or expression, 87, 164, 323, 324
conditional-or operator, 32, 138, 164, 174
configuration file, 255
console, 244
const, 184
const declaration, 179
const keyword, 271, 279
constant, **4**, 30, 31, 59, 62, 67, 85, 90, 115, 188, 238, 246, 257, 384
constant declaration, **4**, 18, 67, 68, 85, 101, 174, 179, 257, 265, 272, 282, 297
constant expression, 41, 96, 152, 174, 281, 344, 378
constant field, 209
constant variable, 262, 264
constant-time operation, 39, 342, 343
constraint, 77, 78, 82, 83, 97, 122, 283, 289, 304, 305, 321
construct, 1, 26
constructed value, 54
constructor, 30, 31, 61, 70, 72, 85, 114, 115, 119, 121, 130, 131, 137, 187, 189, 193, 194, 204–208, 211, 221, 222, 226, 227, 271, 296, 300, 307, 309, 339, 342–345, 349, 352, 363, 371
constructor body, 131, 187, 222, 224, 226, 227, 271
constructor call, 85, 100, 131, 222, 224, 226, 227, 296
constructor call statement, 119
constructor declaration, 211, 221, 222, 227, 363

constructor keyword, 221, 222
constructor overloading, 309, 339
constructor type, 208
contained expression, 118
container, 54
context, 12, 43, 64, 83, 85, 86, 88, 96, 97, 104, 106–109, 111, 119, 129, 172, 180, 213, 254, 271, 281–284, 297, 299, 301, 307, 329, 384
context-free grammar, **4**
context-free grammar, 2
contiguous memory location, 39
continue statement, 181, 185
contravariance, 80, 292, 294, 295, 305
contravariance pattern, 221
contravariant, 80, 304
contravariant parameter, 304
contravariant parameter type, 305
contravariant return type, 292, 305
contravariant type parameter, 321
control, 178, 219, 373, 374
control transfer, 184, 185, 187, 189
conversion, 31, 32, 43, 46, 50, 71, 85–91, 126, 140–144, 149, 153–155, 160, 162, 165–167, 169, 235, 238, 294, 344
converted type, 155, 160
convertibility, 141, 142, 162
convertible type, 142, 146, 152, 155, 160
coroutine, **4**, 94, 326, 327, 329
covariance, 80, 292, 294, 295, 304
covariance pattern, 221
covariant, 80
covariant parameter type, 305
covariant return type, 292, 304, 305
covariant type parameter, 321
creation, 229
creation expression, 100
cross-platform development, 2
curly brace, 108, 170
cyclic dependency, 66

D

data member, 208
data type, 57
deadlock, 322
deallocation, 2
decimal, 14, 18, 86
decimal form, 31
decimal number, 18
declaration, 22, 45, 50, 59, 61–64, 66, 68, 101, 183, 184, 197, 204–206, 212, 215–217, 219, 229, 230, 233, 234, 247, 249–251, 257, 258, 262–265, 268, 271, 272, 280, 282, 294, 345, 357, 359, 364, 367, 374, 377–380, 382–384, 386, 390
declaration annotation, 383
declaration body, 179
declaration name, 263
declaration scope, 60, 193, 205, 231, 250–252, 255, 307
declare, 271
declared class member, 193
declared entity, 60, 378
declared function, 101
declared interface, 230
declared name, 262
declared type, 119, 271, 279
declaring class, 207
decrement, 129, 174
decrement expression, 140–142
decrement operator, 29–31, 95, 96, 139–141, 174
decrementation, 140, 142
default, 79, 109, 268
default constructor, 227
default export directive, 258, 262
default implementation, 278, 364
default keyword, 251, 278
default method, 262, 364
default target, 262
default type, 77, 79, 283
default value, 57, 66, 71, 112, 210, 211, 217, 273, 322, 344, 372
definite assignment assertion, 211
delimiter, 2
denormalized number, 31
denormalized value, 31
derived class, 62, 195, 207, 241, 303, 304, 364, 366, 367
derived interface, 364
derived type, 49, 294
difference type, 293
dimension expression, 344, 345
direct call expression, 119
direct extension, 229
direct implementation, 229
direct subclass, 195, 196, 322
direct subinterface, 230
direct superclass, 195, 205, 227
direct superinstance, 205
direct superinterface, 44, 197, 227, 230, 231, 234, 289
direct supertype, 284, 289
directive, 249, 263, 264
distinct argument, 83
distinct generic declaration, 83
distinct type, 52, 53
distinguishable declaration, 60, 61
distinguishable function, 62
distinguishable functions, 61

dividend, 148–150
division, 148, 149
division operator, 30, 148, 149
divisor, 148–150, 190
do statement, 182, 185, 323
dot operator, 63
dot-separated name, 100
double, 28, 31, 57
double infinity, 91
double NaN, 91
double quotes, 19
double type, 91
dual semantics, 36
dynamic dispatch, 346
dynamically created object, 2
dynamically dispatched overriding, 2

E

EAWorker, 330
effective type, 134, 319, 321
element type, 39
else-block, 181
embedded expression, 170
embedded namespace, 246–248, 280
empty body, 218, 363
empty string, 323
enclosing context, 180
enclosing statement, 185
end marker, 3
ensure-not-nullish expression, 50, 137
entity, 59, 62, 63, 73, 77, 100, 101, 232, 244, 245, 249, 251, 252, 255, 264, 268, 271, 279, 280, 297, 307, 322, 352–355, 369, 374, 378, 380, 463
entity declaration, 62
entry point, 268, 269
entry point function, 268, 269
enum, 235, 257
enum declaration, 64
enum member, 63, 237
enum type, 33
enumeration, 27, 28, 195, 235, 321, 384, 385
enumeration base type, 157, 321
enumeration declaration, 27, 235
enumeration member, 238, 239
enumeration method, 238
enumeration relational operator, 157
enumeration type, 23, 25, 28, 57, 86, 90, 157, 158, 235, 238, 239, 378
enumeration type declaration, 271, 279
environment, 191
environment variable, 255
equality, 36, 149, 159
equality expression, 87, 158, 160, 161, 174
equality operator, 29, 95, 158, 160, 161, 174
erased type, 391
error, 30, 31, 34, 97, 99, 100, 148, 150, 153, 173, 178, 179, 189, 191, 211, 227, 241, 242, 266
error handling, 241
error object, 97, 189
error situation, 242
escape, 20
escape sequence, 19, 20, 340
evaluation, 93, 94, 96, 97, 99, 100, 103, 114, 115, 120, 121, 123, 126, 128, 132, 133, 135, 137, 138, 140, 142, 148, 158, 160, 163–169, 171–174, 178, 181–183, 187, 188, 210, 345, 346
evaluation result, 262
exception, 322, 390
exclusive OR operator, 162
executable code, 214, 215, 271
execution, 97, 103, 114, 173, 178–180, 182, 185, 187, 188, 190, 191, 216, 224, 227, 241, 268, 319
execution path, 76, 224
execution transfer, 188
exit condition, 185
explicit call, 348
explicit class method overload, 356
explicit constructor call, 226
explicit function overload, 257, 273, 354
explicit initialization, 57
explicit instantiation, 77
explicit method overload, 357
explicit overload, 103, 357, 359–361
explicit overload declaration, 124, 307
explicit type, 237, 262
exponent, 151
exponentiation, 95, 151
exponentiation operator, 31
export, 63, 245, 247, 249, 251, 252, 257, 258, 262–264, 268, 381
export annotation, 382
export default, 382
export directive, 262–264, 381
export function, 248
export keyword, 378, 381
export modifier, 257, 463
export namespace, 249
export target, 262
export type, 254, 264, 382
exported declaration, 247, 257, 262
exported entity, 59, 63, 101
exported interface, 262
expression, 3, **4**, 23, 36, 55, 62, 71, 85–88, 93, 94, 96, 97, 99, 100, 103, 106–109, 113–115, 119, 123–125, 128–131, 133–135, 137–142, 144, 147, 152–155, 163, 164, 167–170, 172, 174, 178, 180–184, 187–189, 210, 215, 222, 226, 235, 262, 266, 281–284, 296, 297, 299, 323, 328,

344, 345, 348, 352, 364, 369, 374, 378, 379, 381
expression evaluation, 97
expression statement, 178, 266
expression type, 86, 91, 97, 123, 130, 162, 187, 299, 323
expression value, 85, 182, 188
extended conditional expression, 144, 169, 182, 323
extended equality, 161
extended exponent, 165
extended semantics, 164, 323
extends clause, 195, 230, 363
extends graph, 195, 230
extends keyword, 78
extends Object clause, 195
extension, 229, 230, 254, 363
extension clause, 224, 286

F

factory, 351
factory parameter, 350
field, 47, 48, 52, 59, 85, 109, 111, 114, 120, 121, 126, 128, 165, 168, 193, 204–206, 208–214, 219, 220, 229, 231, 232, 380, 381, 383, 384
field access, 120, 121, 129, 168, 210, 321
field access expression, 50, 120, 121, 128, 210
field annotation, 381
field declaration, 85, 206, 208–210
field initialization, 210, 213, 382
field initializer, 194, 210, 211, 224, 227
field modifier, 208
field name, 56
field overriding, 213
field type, 109, 165
field value, 211, 219
field with late initialization, 208, 211, 224
file, 254, 389
file path, 254
file system, 254, 255, 389
final class, 194, 339, 363
final keyword, 363
final method, 217, 219, 339, 363
final modifier, 194, 195, 215–217
finally block, 190
finally clause, 189, 190
finite value, 148, 149, 153, 155
fit into (*v.*), **4**
fixed array type, 23
fixed-size array type, **4**
fixed-size array, 38, 97, 284, 293, 342
fixed-size array argument, 74
fixed-size array type, 33, 74, 293, 342
flexibility, 346, 364
float, 28, 31, 57, 89
float infinity, 91
float NaN, 91
float type, 91
float zero, 91
floating-point addition, 153
floating-point arithmetic, 153
floating-point calculation, 99
floating-point comparison, 155
floating-point division, 148, 149
floating-point equality test, 160
floating-point expression, 32
floating-point infinity, 91
floating-point literal, 18, 281
floating-point number, 30, 31, 143
floating-point operand, 91, 149, 153, 156, 160
floating-point operation, 31, 148, 149, 153
floating-point operator, 31
floating-point remainder, 150
floating-point remainder operation, 149, 150
floating-point subtraction, 153
floating-point type, 28, 30, 31, 39, 86, 89, 91, 148, 149, 153, 344
floating-point value, 31, 143, 155, 160
floating-type multiplication, 147
flush to zero, 31
folder, 254, 389
for statement, 180, 183, 185, 323
for-of loop, 183
for-of statement, 180, 183, 185, 347
for-of type annotation, 184
for-of type statement, 352
for-variable, 352
form feed, 10, 20
formal parameter, 85, 171
function, 27, 34, 42, 59, 61, 62, 64, 66, 70, 72–79, 81–83, 85, 101, 115, 119, 124, 159, 161, 180, 185, 187–190, 195, 246, 257, 266, 268, 271, 282, 296, 297, 299, 307, 339, 345, 346, 352–355, 362, 364, 368, 371, 372, 374, 380, 385, 390
function body, 69, 75, 172, 187, 269, 271, 273, 327, 362
function body declaration, 63
function call, 40, 41, 50, 62, 69, 71, 76, 85, 96, 100, 123, 124, 129, 296, 307, 321, 353, 364, 367, 371, 372
function call expression, 76, 123, 124
function declaration, **4**, 63, 69, 171, 257, 354, 364
function increment, 207
function name, 63, 345, 364
function object, 44, 161
function overloading, 339
function parameter, 66, 75
function reference, 101, 355

function return type, 297, 299
function scope, **4**, 63
function signature, 42, 101
Function type, 44
function type, 23, 25, 26, 33, 42–44, 81, 123, 158, 171–173, 292, 305, 364, 368, 371, 372, 390
function type equality operator, 161
function type parameter scope, **4**, 63
function type with receiver, 25, 368–370
function with receiver, 119, 121, 257, 364–368, 385
function with receiver declaration, 364
functional object, 124
functionality, 326, 364

G

general import, 254
generic, 2, **4**, 65, 77, 79, 82, 83, 97, 230, 294, 304, 319, 353, 368, 390
generic class, 27, 78, 80, 81, 194, 242, 289, 346, 391
generic class declaration, 197
generic declaration, 77, 78, 82, 83, 230
generic entity, 81
generic function, 69, 78, 83, 101, 366
generic instantiation, 77, 78, 81–83, 101, 102, 132, 230, 297, 353, 354
generic interface, 78, 80, 197, 230, 289, 346
generic method, 83, 101, 102
generic parameter, 77
generic tuple, 321
generic type, **4**, 27, 35, 64, 65, 79, 80, 83, 113, 132, 289, 294, 321
genericity, 2
get-accessor, 220, 373
getter, 52–54, 219, 220, 231–234, 299, 373, 374
getter body, 220
getter parameter, 221
getter return type, 299
goal symbol, 2, 3, **5**
gradual underflow, 31
grammar, **5**
grammar production, 3
grammar rule, 3

H

hard keyword, 12
header, 216
hexadecimal, 14, 20
hidden field, 204
hidden member, 229
hiding, 229
high-level language, 2
high-level sequence, 224
horizontal tab, 20
horizontal tabulation, 10
host system, 389

I

identifier, 2, 10–13, 26, 27, 59, 70, 71, 108, 109, 120, 121, 128, 181, 185, 187, 194, 208, 215, 219, 221, 230, 233, 251, 252, 264, 356, 357
identity, 294
identity conversion, 88
IEEE 754, 30, 31, 89, 91, 147–150, 153, 155, 160
if statement, 180, 181, 323
immutable variable, 180
implementation, 31, 159, 193, 195, 203, 204, 213, 214, 216–218, 220, 222, 229, 230, 232, 233, 255, 275, 277, 280, 286, 299, 300, 339, 346, 347, 360, 361, 363, 364, 383, 389
implementation clause, 286
implementation method, 218
implementing, 193
implements clause, 197
implicit conversion, 18, 170, 294
implicit function overloading, 307
implicit method overloading, 308
import, 63, 244, 251, 253–258, 262, 264, 266, 354, 381, 383
import annotation, 382
import binding, 249, 251, 252, 254, 255, 258
import declaration, 249, 255
import default, 382
import directive, 244, 249, 250, 253–255, 257, 263, 381
import outcome, 255
import path, 249, 251, 253–255, 265, 389
import type, 250, 254, 382
import type directive, 254
imported declaration, 244
imported entity, 101
imported file, 255
imported function, 101
imported module, 266, 381
in keyword, 80
in-place type declaration, 25
in-position, 80
in-variance, 77
inclusive OR operator, 162
incompatibility, 342
increment, 129, 174, 296
increment expression, 140, 141
increment operator, 29–31, 95, 96, 139–141, 174
incrementation, 140
index, 55, 342–344
index expression, 41, 55, 125–128
index parameter, 346
index subexpression, 166–168

indexable type, 39, 125, 275, 345
indexing, 36, 100, 127, 275
indexing expression, 50, 55, 100, 125, 127–129, 166–168, 345, 346
indexing expression evaluation, 346
indistinguishable type, 294
inference, 66
inference type, 112
inferred interface type, 112
inferred type, 67, 68, 76, 83, 96, 97, 104, 106–108, 111, 133, 146, 172, 183, 184, 220, 269, 281, 283, 297, 299, 347
infinite operand, 153
infinite value, 153
Infinity, 151
infinity, 91, 99, 143, 148–150, 153, 155
infinity double, 91
inheritance, 2, 33, 205, 208, 209, 214–217, 221, 227, 229–231, 234, 300, 339, 348, 357, 358, 360, 364, 378
inherited class member, 193
inherited field, 214
inherited member, 205, 227, 229
initial value, 40, 41, 85, 114
initialization, 53, 57, 66, 103, 113, 114, 130, 131, 173, 180, 183, 204, 209–211, 213, 221, 222, 224, 266, 271, 282, 322, 345, 382, 383, 390
initialization expression, 107
initializer, 66, 68, 109, 210, 213, 247, 248, 271, 274, 282, 284, 297, 321, 378
initializer block, 120, 193, 205, 210, 245
initializer declaration, 274
initializer expression, 66–68, 103, 210
innermost declaration, 63
instance, 33, 34, 55, 63, 100, 107, 114, 130, 131, 160, 161, 171, 173, 183, 184, 209, 210, 216, 217, 221, 224, 229, 239, 342, 344, 347, 351, 370
instance creation expression, 107, 173, 222, 309
instance entity, 62
instance field, 121, 208, 209, 211, 213, 222, 224
instance field access, 121
instance field initializer, 210
instance member, 59, 62, 193
instance method, 102, 119, 121, 123, 124, 216, 218, 226, 234, 301, 349, 351, 356, 363, 366
instance method call, 119
instance name, 209
instance own field, 222
instance variable, 229
instanceof expression, 131, 132, 319, 390
instanceof operator, 131
instantiated generic type, 132
instantiation, 27, 35, 77–83, 101, 130, 131, 194, 197, 209, 229, 289, 308, 321, 348–351
int, 16, 28–30, 57, 89
int type, 91, 238, 268, 344
integer, 14, 16, 18, 31, 36, 37, 39, 147, 150, 153, 342, 344
integer addition, 99, 153
integer arithmetic, 153
integer bitwise operator, 29, 31, 162
integer division, 97, 99, 148, 154
integer equality test, 160
integer expression, 269
integer literal, 16, 281, 283
integer multiplication, 99, 147
integer number, 342, 343
integer operand, 148, 149, 156, 160
integer operator, 30
integer overflow, 148
integer remainder, 97, 99, 149
integer subtraction, 153
integer type, 28, 30, 86, 89–91, 125, 126, 144, 154, 162, 237, 299, 323, 344
integer value, 18, 29, 143, 148, 153, 237, 344
interface, 27, 33, 50, 59, 63, 77, 78, 80–82, 101, 102, 111–114, 119, 121, 122, 195, 197, 203–205, 212, 213, 217, 220, 229–234, 257, 277, 289, 303, 308, 322, 345–347, 358–361, 364, 371, 378, 380, 385
interface body, 234, 364
interface declaration, 27, 64, 229–231
interface field, 85
interface getter, 385
interface inheritance, 234
interface keyword, 378
interface level scope, **5**, 63
interface member, 62, 63, 231
interface method, 220, 233, 278, 352, 359, 360, 385
interface method declaration, 233, 364
interface name, 230
interface property, 112, 231–233
interface setter, 385
interface type, 23, 25, 27, 29, 33, 45, 47–49, 52–54, 59, 83, 108, 111, 112, 119, 133, 197, 229–231, 286, 304, 365, 368
interface type declaration, 289
interface type variable, 59
internal module, 463, 464
invariance, 80, 294, 295, 304
invariant, 80
invariant type parameter, 321
invocation, 322, 348, 349, 351
iterable class, 183, 347, 348
iterable class instance, 277
iterable interface, 183, 347
iterable type, 115, 183, 184, 347, 358
iteration, 183–185

iterator, 115, 184, 347, 348

K

key, 54, 113, 128, 166, 168
key type, 113, 114, 127
key-value pair, 166, 168
keyof keyword, 48
keyof type, 48, 49
keyword, 2, **5**, 10, 12, 13, 18, 21
keyword super, 62
keyword this, 62
keyword undefined, 35

L

label, 181, 185
label identifier, 185
lambda, 44, 72, 76, 100, 172–174, 181, 187, 292, 297, 299, 368, 371, 372, 385
lambda body, 119, 171–173, 187, 299, 371
lambda call, 85, 172
lambda expression, 76, 100, 119, 123, 171–174, 181, 297, 327, 364, 370, 380
lambda expression call, 172
lambda expression type, 172
lambda expression with receiver, 93, 94, 364, 370–372, 380
lambda function, 344
lambda parameter, 172, 371
lambda return type, 172, 297, 299
lambda signature, 171–173
lambda with receiver, 368
language element, 377
late initialization, 211
lazy operator, 138
left shift, 154
length property, 342, 343
let, 184
let declaration, 179
lexical element, 2
lexical grammar, 2, 3
lexical input, 9, 10
lexical input element, 10
lexical notation, 2
lexical structure, 2
line separator, 2, 9, 10, 22, 377
line separator character, 10
linearization, **5**, 47
linkage, 97
literal, **5**, 10, 14, 16, 18, 21, 35–37, 54, 55, 97, 100, 104, 114, 127, 128, 174
literal expression, 106
literal type, 21, 23, 25, 33, 37, 38, 46, 47, 57, 68, 114, 127, 289, 321
literal value, 18, 46
local declaration, 380
local variable, 88
logical complement, 144, 323
logical complement expression, 144
logical complement operator, 144
logical expression, 87, 162, 163
logical operator, 32, 95, 162, 163, 174
long, 16, 28–30, 57, 89
long type, 37, 91
lookup sequence, 389
loop, 181, 183–185, 324
loop body, 181, 184
loop index variable, 183
loop iteration, 183, 185
loop label, 181
loop scope, 184
loop statement, 181, 184, 185, 187
loss of information, 31, 86, 148, 149, 153
loss of precision, 150
low-level representation, 2
low-order bit, 147, 153
lowest-order bit, 154
lvalue, 96

M

magnitude, 149, 150, 153
mandatory call, 224
mandatory parameter, 43
mantissa bit, 31
mapped value, 128
mask value, 154
match, 188
match (*v.*), **5**
member, 62, 205, 206, 216, 229
member access, 95
member field, 120
memory location, 39, 342
meta-annotation, 383, 384
metadata, 377, 384
metasymbol, 3, **5**
method, **5**, 30, 31, 33, 34, 37, 39, 44, 47, 48, 51–54, 59, 62, 66, 70, 72, 74–78, 81–83, 85, 94, 101, 103, 112, 115, 122, 123, 126, 131, 180, 185, 187, 189, 193, 204–208, 210, 214–218, 220, 221, 229–231, 233, 234, 238, 239, 271, 278, 295–297, 299, 300, 303, 304, 307, 308, 339, 347–352, 356–364, 366, 371, 372, 380, 390, 391
method body, 75, 76, 119, 172, 187, 195, 216, 218, 226, 233, 271, 299, 347, 363, 364
method body declaration, 63
method call, 40, 41, 50, 70, 71, 85, 86, 96, 100, 121–123, 129, 189, 215, 275, 296, 308, 321, 348, 364, 369, 371, 372, 390

method call expression, 76, 100, 121, 123, 131, 215, 364
method declaration, 195, 206, 214–218, 221, 231, 327, 352, 363, 364
method implementation, 112
method member, 206
method modifier, 123, 215, 219, 357
method name, 63, 215, 233, 361
method overload, 204
method overloading, 339
method overriding, 300
method parameter, 66, 75
method parameter name, 63
method reference, 101, 102, 161, 361
method return type, 123, 297, 299
method scope, **5**, 63
method signature, 102, 217, 218, 233, 300
migration, 2
mode of evaluation, 178
modification, 184
modifier, 62, 184, 194, 195, 209, 212, 216, 217, 250
modifier async, 273
modifier const, 184
modifier declare, 271
modifier export, 262
modifier let, 184
modifier static, 209
modular programming, 2
module, 2, 50, 62, 63, 101, 244, 246, 247, 249–251, 254, 255, 257, 262–266, 268, 271, 280, 321, 322, 354, 367, 389, 390, 463
module header, 244
module initialization, 390
module level scope, **5**, 62
module name, 244
module visibility, 463
module with ambient declaration, 262
multi-targeting, 2
multifile module, 267
multiline comment, 22
multiline string, 19–21, 170
multiline string literal, 20
multiplication, 147, 148, 154
multiplication operator, 148
multiplicative expression, 31, 87, 146, 174
multiplicative operator, 29, 31, 146, 174
mutable variable, 180

N

name, 59, 60, 62–64, 66, 69, 75, 101, 239, 251, 252, 255, 256, 258, 268, 303, 307, 348, 355, 358, 361, 366, 374, 377
name binding, 254
name scope, **5**
name-value pair, 108
named class, 114
named entity, 60
named function, 69
named member, 235
named reference, 100–103, 322
named store location, 66
named type, 25, 27
named variable, 168
namespace, 2, 62, 101, 245–249, 257, 279, 321, 322, 352, 354, 385
namespace declaration, 62, 245, 257, 280, 321
namespace level scope, **5**, 62
namespace name, 247, 280
namespace scope, 354
namespace variable, 245
NaN, 31, 91, 143, 148–151, 153, 155, 156, 158, 160, 324
Nan, 91
NaN value, 99
native, 357
native accessor, 373
native constructor, 222, 363
native function, 69, 299, 327, 339, 362, 373
native keyword, 362, 363
native method, 217, 218, 339, 362, 363
native modifier, 217, 219, 373
nearest value, 31
negation, 143, 153
negative infinity, 91, 155, 160
negative integer, 148, 149
negative integer value, 344
negative zero, 153, 155, 160
nested literal, 103
nested multiline string, 170
nested namespace, 280
nested statement, 63, 187
nested union type, 47
nesting, 56
never, 24
never type, 33, 34, 47, 49, 76, 133, 299
new expression, 349
newline, 20
newline character, 10
no-argument return statement, 327
no-break space, 10
non-generic, **5**
non-generic type, **5**
non-abstract class, 194, 195
non-abstract instance method, 217
non-abstract method, 216, 217
non-abstract subclass, 217
non-alias, 47
non-aliased type, 28
non-ambient interface, 278

non-ambient method, 275
non-boolean type, 323
non-class type, 195
non-empty body, 363
non-empty string, 323
non-explicit type, 237
non-exported declaration, 247
non-generic class, 82, 286
non-generic entity, 81
non-generic function, 82
non-generic interface, 82
non-generic method, 82
non-generic type, 27
non-generic type alias, 82
non-initialized variable, 66
non-interface type, 230
non-native accessor, 373
non-native constructor, 222
non-negative integer number, 39
non-nullish type, 47, 49, 120, 129, 138
non-nullish variant, 137
non-nullish-type, 78
non-numeric type, 126
non-optional field, 204
non-optional parameter, 118
non-optional property, 112
non-relative import path, 254, 255
non-relative path, 255
non-standalone expression, 96
non-static class, 206
non-static entity, 205
non-static field, 194, 208–210, 220, 226
non-static field declaration, 210
non-static member, 205
non-static method, 216
non-static modifier, 212, 362
non-string operand, 86, 283
non-union type, 46
nonterminal, 2, 3, **6**
nonterminal symbol, 3, **6**
nonzero, 323
nonzero double, 91
nonzero operand, 153
nonzero value, 32
normal completion, 97, 99, 100, 114, 126, 128, 141, 142, 165–168, 173, 174, 178, 189–191, 218, 241
normal execution, 241
normal method call, 348
normalization, 45–47, 299
normalized union type, 46, 47
notation, 71, 206, 209, 232, 248, 371, 381, 382, 384
notion, 211
null, 24, 26, 57, 129, 137, 161
null literal, 21, 35
null pointer dereferencing, 241
null reference, 21
null safety, 49
null type, 33, 35, 37, 52, 289
null-coalescing operator, 95
nullable type, **6**
nullish expression, 324
nullish object reference, 120
nullish type, 21, 35, 49, 50, 86, 123, 125, 129, 137, 138, 211, 323
nullish value, **6**, 49, 50, 121, 123, 125, 129, 137, 138
nullish-coalescing assignment, 167
nullish-coalescing expression, 50, 138
nullish-coalescing operator, 138
nullish-safe option, 49, 50
nullish-type, 78
number, 2, 18, 55, 57, 323, 361, 362
number type, 30, 291
numeric base type, 90
numeric casting, 140
numeric casting conversion, 91
numeric constant expression, 297
numeric context, 87
numeric conversion, 87, 89, 91, 158, 160
numeric equality, 160
numeric equality operator, 29, 31, 158, 160
numeric expression, 324
numeric literal, 2, 133
numeric literal type, 47
numeric operand, 153
numeric operation, 31
numeric operator, 29, 31, 85, 283, 284
numeric operator context, 283
numeric promotion, 30, 31
numeric relational operator, 29, 31, 155, 157
numeric type, 28–31, 37, 47, 54, 87–89, 91, 107, 126, 133, 140, 142, 144, 146, 148–150, 152–156, 160, 162, 235, 237, 283, 299, 378
numeric type operand, 153
numeric types conversion, 31, 87, 126, 140, 142, 146, 148, 149, 153, 155, 156, 160, 162
numeric value, 89, 91, 158
numeric widening, 142
numeric widening conversion, 153

O

Object, 24, 28, 189, 195, 205, 229, 231, 286, 289, 296
object, 2, 33, 36, 40, 41, 67, 78, 97, 119, 121, 130, 152, 188, 216, 221, 229, 230, 284, 344, 347, 363
object field, 224
Object literal, 297
object literal, 53, 55, 85, 107–114, 119, 133, 281, 379, 381

object literal expression, 108
object orientation, 2
object reference, 120, 122, 357, 360, 364
object reference expression, 125, 126, 128
object reference subexpression, 168
Object type, 47
object type, 300
object-oriented, 2
object-oriented programming (*OOP*), 339
octal, 14
OOP (*object-oriented programming*), 2, 339
operand, **6**, 29–31, 86–88, 91, 99, 131–133, 140, 142–144, 146–158, 160, 161, 163–169, 281, 283
operand expression, 147, 153, 162, 168, 169
operand null, 86
operand string, 152
operand type, 154, 155, 281
operand value, 142, 144, 147, 153, 162, 163
operation, **6**, 31, 37, 143, 153, 319, 322
operation overflow, 153
operation sign, **6**
operation type, 38
operational function, 372
operator, 10, 13, 21, 30, 31, 39, 41, 95, 97, 99, 125, 131, 134, 137, 138, 141, 142, 144, 149, 152, 154, 156–158, 160, 162, 165, 188, 296
operator (*in programming languages*), **6**
operator context, 152
operator evaluation, 97
operator precedence, 95, 99
operator sign, 2
operator undefined, 86
operator validity test, 28
optional annotation, 171
optional arbitrary code, 224
optional argument, 372
optional field, 204, 208, 210, 211
optional identifier, 221
optional name, 221
optional parameter, 43, 70, 71, 73, 171, 220, 273, 292, 344, 372, 380
optional property, 52, 112, 232, 233
ordinary function, 346
ordinary method, 347, 358, 360, 391
original variable, 173
out keyword, 80
out-variance, 77
overflow, 30, 31, 143, 147–150, 153, 154
overlap, 63
overlapping, 160
overload, 205, 275, 353, 357, 361
overload declaration, 339, 352–354, 357, 358
overload equivalence, 308, 309
overload keyword, 352
overload resolution, 122, 124, 353
overload set, 307, 315, 352, 353
overload-equivalence, 307, 308
overload-equivalent signature, 307–309
overloaded entity, 352, 353, 361
overloaded function, 252, 307, 308, 354, 355, 357
overloaded interface, 359
overloaded method, 308, 357, 358, 360–362
overloading, **6**, 61, 268, 284, 300, 307, 352, 391
overloading signature, 215
overridden field, 213
overridden member, 229
overridden method, 218, 361, 363, 391
override, 214, 360
override compatibility, 304, 305
override keyword, 212
override method, 295
override modifier, 195, 216, 217
override-compatibility, 300, 301, 303
override-compatible signature, 221, 227, 230, 234, 294, 300, 301, 303, 304
overriding, 2, 94, 193, 212, 213, 215, 217, 218, 221, 227, 229, 234, 283, 284, 289, 295, 300, 301, 303, 305, 306, 346, 357, 358, 361, 363, 364, 391
overriding entity, 294
overriding field, 212, 214
overriding method, 217, 219, 391
own (*adj.*), **6**
own field, 208

P

package, 463
paragraph separator character, 10
parallel execution, **6**
parallel-run coroutine, 326
parameter, 34, 70, 75, 101, 118, 119, 124, 168, 172, 219, 220, 227, 232, 268, 273, 292, 294, 296, 304, 305, 344, 349–351, 364, 365, 368, 370–373, 380, 385, 390
parameter constraint, 283
parameter declaration, 171
parameter list, 42, 70, 72
parameter name, 43, 63, 70, 72, 368
parameter type, 43, 71, 221, 292, 293, 295, 297, 306, 321
parameter with default value, 71
parameterization, 77, 78, 83
parameterized class type, 197
parameterized declaration, 82
parameterized interface, 230
parameterized type, 230
parameterless constructor, 109, 114, 344
parameterless function type, 350

parameterless method, 347
parent class, 300
parentheses, 13, 26, 99, 100, 131, 165, 166, 381
parenthesized expression, 118, 174
path, 241, 389
path component, 254
path mapping, 255
path rule, 254
pattern, 94
permutation, 294
platform API, 279
platform-dependent code, 363
point of declaration, 63
policy, 384
polymorphism, 352
polymorphism by name, 307, 352
positive infinity, 91, 155, 160
positive zero, 153, 155, 160
postfix, 29, 31, 129, 137, 139, 140
postfix operator, 95
precedence, 26, 63, 95, 99, 158
predefined constant, 30, 31, 340
predefined constructor, 30, 31, 340
predefined method, 30, 31, 340
predefined numeric types conversion, 30, 31, 144
predefined operator, 97
predefined reference type, 19
predefined type, 23, 24, 27, 28, 44, 46, 87, 141
predefined type declaration, 49
predefined value type, 167
prefix, 16, 18, 29, 31, 41, 72–74, 129, 137, 141, 142, 174, 271, 279, 377, 378
prefix operator, 95
prefix readonly, 26, 40, 70, 71
primary constructor, 222, 224, 226
primary expression, 119, 125, 129
primary name, 24
private, 205–207, 227, 357
private field, 56, 204, 220
private member, 195, 207, 365
private method, 196, 234, 300, 303, 364
private modifier, 217
production, 2, 3, **6**
program completion, 337
program entity, 2, 77
program entry point, 266, 268
promise, 51
promise object, 329
promoted operand value, 142
promoted type, 153, 154
promoted value, 143
promoting, 153
promotion, 143, 146, 154
propagation, 191
property, 44, 52–54, 110, 112, 113, 203, 204, 209, 229, 231, 232, 234
property length, 39
property type, 232
protected, 205–208, 357
protected member, 365
protected modifier, 207
provably distinct instantiation, 83
proxy, 329
pseudogeneric static method, 390
public, 205, 206, 208, 357
public member, 365
public method, 231
public modifier, 208, 227
punctuator, **7**, 10, 13

Q

qualification, 209, 236, 329
qualified access, 59, 63
qualified form, 249
qualified import, 62
qualified name, **7**, 59, 62, 100, 101, 124, 205, 206, 209, 245, 248, 251, 256, 354, 381, 463
qualified type name, 27, 29
qualifier, 245

R

radix, 14, 18
re-assignment, 49
re-export, 262, 264, 265
re-export declaration, 264
re-export directive, 244, 264
re-exported declaration, 265
re-exporting declaration, 262
read permission, 241
readability, 18
readonly, 41, 47, 80, 97, 121, 203, 209, 226, 232, 233
readonly array, 165, 166
readonly array type, 40
readonly field, 120, 209, 385
readonly modifier, 209
readonly name, 386
readonly parameter, 71
readonly property, 56
readonly tuple, 165, 166
readonly type, 26, 54
reassignment, 54
receiver, 364, 367
receiver parameter, 368, 372
receiver type, 365–368
record, 125
record access expression, 166, 168
record element, 168

record indexing expression, 101, 128, 166, 168
record instance, 128, 166, 168
Record type, 55
record type, 113, 125, 127, 133
record utility type, 54
recursive reference, 65
recursive replacement, 28
reference, 33, 229
reference expression, 125, 126
reference subexpression, 165, 166
reference type, 33, 36, 41, 45, 49, 57, 59, 86, 120, 126, 128, 165, 167, 193, 194, 229
reflexive closure, 284
rejected promise, 337
rejection handler, 337
relational expression, 87, 155
relational operator, 32, 36, 95, 155–158, 174
relative import path, 254, 255
relative location, 255
relative path, 255
remainder operation, 149, 150
remainder operator, 30, 31, 97, 149, 150
renaming, 262, 263, 382
repeatable annotation, 380
replacement, 28
required parameter, 70
required property, 53, 56, 232
reserved word, 12
resizable array, 39, 100, 130, 284, 342, 344
resizable array type, 7, 38, 39
resolution, 159, 254, 255, 328
resolving, 255
rest parameter, 43, 70, 72–74, 118, 292, 296
restriction, 54, 55, 127, 210, 275–277, 374, 384
retention annotation, 383, 385
retention policy, 385
return, 34, 179, 190
return expression, 187
return statement, 76, 172, 179, 187, 218, 226, 266, 268, 299, 327
return type, 34, 42, 70, 76, 123, 124, 171, 172, 179, 187, 206, 218, 220, 221, 232, 268, 273, 277, 292, 297, 299, 304, 305, 327, 347, 373
right shift, 154
root coroutine, 329
round to nearest, 31
round toward zero, 31
round-to-nearest mode, 89, 148, 149, 153
round-toward-zero, 91
rounding, 89, 148, 149, 153
rounding mode, 31
rounding rule, 91
rounding rules, 91
rounding toward zero, 31
routine, 149, 321
runtime, 36, 39, 49, 97, 115, 119–121, 126, 134, 140, 142, 144, 152, 154, 160, 163–165, 167, 172, 174, 190, 191, 210, 211, 300, 339, 342, 343, 345, 384, 385, 390, 463
runtime check, 134, 293
runtime error, 39, 42, 44, 88–91, 97, 125, 133, 134, 144, 148–151, 190, 211, 241, 278, 318, 321, 342, 344, 382
runtime evaluation, 173, 322, 345
runtime expression, 339
runtime implementation, 390
runtime polymorphism, 300
runtime system, 242, 293

S

safe field access, 50, 120
safe function call, 50
safe indexing expression, 50
safe method call, 50, 121
safe operation, 50
safety, 322
scope, 59, 62–64, 77, 171, 181, 191, 193, 194, 205, 230, 231, 250, 255, 265, 268, 322, 354, 374
secondary constructor, 222, 226
selective binding, 252, 263
selective export, 263
selective export directive, 263
selective import, 251
semantic check, 123, 124, 132, 227, 234, 254, 264, 296, 301
semantic correctness check, 123
semantic equivalent, 232
semantic rule, 281, 283
semantic term, 281
semantics, 48, 160, 161, 164, 210, 226, 266, 273, 275, 283, 284, 304, 319, 323, 324, 329, 352, 353, 377, 383
semi-automatic transition, 2
semicolon, 13, 22, 218, 363, 372
separator, 3, 59, 169, 372
sequence, 3, 36
set of functions, 354
set of methods, 356
set of values, 340
set-accessor, 220, 373
setter, 52–54, 110, 168, 219, 220, 231–234, 373, 374
setter parameter, 221
shadow parameter, 171
shadowing, 75, 171
shared memory, 326
shift, 154
shift distance, 154
shift expression, 31, 87, 154, 174

shift operation, 154
shift operator, 95, 154, 174
short, 28, 29, 57, 89, 91
shortcut notation, 248
side effect, 96, 147, 153, 158, 162–164
sign-extension, 154
signature, 42, 47, 48, 60–62, 69, 70, 216–218, 233, 234, 269, 294, 300, 303, 304, 307–309, 345–347, 349, 350, 353, 359, 368, 374, 391
signed infinity, 31, 148, 149, 153
signed integer comparison, 155
signed right shift, 154
signed shift operator, 29, 31
signed zero, 31
simple assignment operator, 99, 165
simple name, **7**, 59, 62, 100, 101, 252, 268
simple type name, 27, 29
single quote, 19, 340
slash character, 254
smart compiler, 297
soft keyword, 12, 13
source, 254, 384
source code, 10, 23, 384, 391
source file, 389
space, 10, 377
space allocation, 345
special character, 2
spread, 118
spread expression, 73, 74, 115, 124, 296
spread operator, 72–74, 296
square bracket, 13, 41
standalone expression, 96, 97, 169, 281–283
standard annotation, 383
standard library, 31, 39, 149, 241, 242, 244, 268
state, 28, 33
statement, 22, 97, 178–183, 185, 187, 189, 191, 265, 266, 268
statement execution, 178
static, 209
static block, 204, 205, 321
static data member, 208
static dispatch, 367
static entity, 62, 205
static field, 120, 208–210, 321, 322
static field access, 120
static initialization, 245, 321, 322
static initializer, 327
static keyword, 363
static member, **7**, 59, 62, 63, 94, 193, 205, 206
static member declaration, 63
static method, 37, 102, 121, 123, 124, 215, 216, 219, 238, 348–350, 356, 390
static method call, 123
static modifier, 212, 216, 217, 362
static override field, 213
static type, 76
statically typed language, 23
sting concatenation, 152
sting literal, 390
sting type, 152
storage, 250, 265, 389
storage management, 2, 389
string, 18, 24, 26, 29, 31–33, 36–38, 44, 47, 82, 85, 97, 125, 126, 128, 152, 158, 167, 170, 183, 188, 232, 246, 290, 296, 321, 323, 346, 361, 362, 372
string comparison, 157
string concatenation, 31, 36, 152, 167, 283
string concatenation operator, 29, 31, 32, 170
string context, 86, 87
string conversion, 86, 87, 152, 283
string element, 127
string indexing, 126
string interpolation, 20
string interpolation expression, 170
string length, 126
string literal, 19, 20, 36, 170, 254, 289, 384
string literal type, 54, 158
string object, 36, 152
string operand, 86
string operator, 283, 284
string operator context, 283
string relational operator, 157
string type, 33, 36, 54, 86, 90, 127, 128, 135, 152, 160, 170, 174, 235, 237, 238, 268, 283, 289, 346, 378, 383
string value, 36, 127, 157
strong typing, 297
struct, 385
struct keyword, 390
structured concurrency, 326
structuring rule, 3
subclass, 119, 194–196, 205, 217, 218, 242, 284, 286, 300, 301, 322, 357, 361, 363, 391
subcomponent (*derived component, child component*), **7**
subexpression, 97, 125, 165, 167
subinterface, 197, 205, 230, 284, 286, 300, 303, 360
subset, 219, 254
substitution, 197, 230
subtraction, 13, 140, 142, 143, 153
subtype, 33, 78, 82, 97, 132, 134, 283, 284, 286, 289–295, 304, 306, 328, 347
subtyping, 38, 43, 47, 68, 78, 80, 119, 132, 134, 283, 284, 289, 290, 292–294, 304, 347, 368
super, 94, 210, 224, 227
super keyword, 172, 210, 216, 226
superclass, 94, 120–122, 193–195, 197, 209, 212–215, 217, 226, 227, 229, 284, 286, 289, 300, 301,

304, 357, 361
superclass accessor, 121
superclass constructor, 213, 227
superclass constructor call, 226
supercomponent (*base component, parent component*), 7
superconstructor, 224
superinstance, 205
superinterface, 193, 197, 203, 209, 212–214, 227, 229, 230, 234, 284, 286, 289, 300, 303, 304, 360
supertype, 38, 57, 68, 107, 284, 286, 289, 291, 292, 295, 300, 306, 319
surrounding class, 371
surrounding context, 297
surrounding function, 180, 185, 187
surrounding interface, 371
surrounding method, 180, 185, 187
surrounding scope, 191
surrounding type, 172
switch expression, 187, 188
switch statement, 184, 185, 187, 188
synchronization, 322, 326, 336
syntactic grammar, 3, 10, 22
syntactic notation, 2
syntactic structure, 2
syntactical form, 130
syntax, 1, 39–41, 43, 72, 76–78, 94, 100, 103, 114, 115, 118–121, 123, 124, 130, 131, 135, 146, 152, 154, 155, 158, 162–164, 170, 174, 178–185, 187, 189, 190, 195, 197, 204, 208, 214, 221, 222, 229, 231, 233, 235, 244, 245, 249, 254, 257, 262–265, 271, 273–277, 279, 328, 340, 344, 354, 356, 364, 368, 371, 372, 378, 379, 381, 382
syntax production, 22

T

target, 385
target annotation, 384
target type, 86, 87, 91, 96, 97, 133, 134, 281, 283, 299, 321
target-specific hardware, 2
taskpool, 330
template, 2
term 2, 338
term1, 338
terminal, 3, 7
terminal symbol, 3, 7, 9, 10
termination, 187
ternary conditional expression, 32, 169, 323
ternary conditional operator, 29, 31, 32, 169, 174
ternary operator, 95
then-block, 181
third-party library API, 279
this, 112, 210, 224
this keyword, 76, 119, 172, 210, 216, 218, 224, 226, 364, 365, 368, 370, 371
this method, 210
this statement, 218
throw, 31, 34
throw statement, 189, 242
thrown object, 189
thrown value, 189
token, 2, 7, 9, 10, 13, 59
tokenization, 7, 10
top-level declaration, 244, 257, 258, 262, 263, 280, 364, 373, 374, 380
top-level function, 390
top-level statement, 69, 187, 244, 249, 265, 266, 268, 280, 321
top-level statements, 268
top-level type, 257
top-level variable, 75, 257
trailing comma, 103, 108
trailing lambda, 121, 371, 372
trailing lambda call, 121, 123
transitive closure, 196, 230, 284
traversing, 347
treemap, 255
truncated number, 31
truncation, 31, 37, 144, 149, 154, 162
truthiness, 323
try block, 189, 191
try statement, 189, 191, 242
try-catch, 190
tuple, 41, 103, 104, 106, 118, 165, 296
tuple argument, 73
tuple type, 23, 25, 26, 41, 71–73, 81, 115, 284, 294, 321
two's-complement format, 147, 153
two's-complement integer, 154
two's-complement representation, 143
two's-complement value, 143
type, 24, 28, 46, 52–54, 56, 57, 59, 67, 68, 70, 73, 82, 85, 96, 97, 100, 104, 106, 107, 109, 118, 122, 126–128, 132, 134, 137–140, 155, 160, 180, 182, 183, 189, 195, 197, 205, 212, 218, 238, 250, 264, 281–284, 289, 290, 292–294, 299, 304, 319, 321, 322, 344–346, 349, 360, 364, 368, 378, 380, 382, 385, 390
type alias, 25, 27–29, 40, 43, 47, 64, 65, 77, 78, 81, 82, 294, 378, 385
type alias declaration, 27, 65, 294
type annotation, 35, 37, 66–68, 76, 97, 123, 124, 171, 232, 297, 352
type argument, 27, 29, 35, 65, 78, 79, 81–83, 101, 103, 197, 230, 321, 353, 390

type boolean, 162
type call expression, 348–351
type cast, 132
type char, 340
type declaration, 23, 41, 45, 49, 64, 179, 257, 266
type enum, 235
type enumeration, 235
type erasure, 132, 134, 319, 321, 390, 391
type for annotation field, 378
type identity, 39, 294
type import, 254
type in parentheses, 25
type inference, 23, 47, 49, 67, 68, 76, 83, 85, 96, 97, 103, 104, 106–108, 111, 133, 146, 171, 172, 220, 282–284, 297, 299, 347, 373
type instantiation, 79
type int, 87, 90, 344
type Iterator, 347
type mapping, 319, 321
type modifier, 250, 254
type name, 27, 29, 37, 132, 348
type null, 21
type of expression, 281
type parameter, 23, 25, 27, 33, 35, 57, 63, 65, 69, 77–83, 122, 131, 132, 137, 171, 206, 283, 289, 294, 304, 305, 321, 344, 353, 365, 368, 391
type parameter declaration, 27
type parameter scope, 7
type parameterized entity, 2
type preservation, 321
type property, 113
type readonly, 54
type reference, 7, 25, 27–29, 78, 132
type safety, 46, 293, 297
type string, 36, 235
type structure, 26
type undefined, 21
type void, 21, 76
type-safe call, 391
typed catch clause, 189
typeof expression, 135
typeof operator, 95
types conversion, 162

U

unary bitwise complement expression, 144
unary expression, 87, 139
unary logical complement expression, 144
unary minus, 142, 143, 174
unary minus operator, 31
unary negation operation, 142
unary numeric expression, 283
unary numeric promotion, 142
unary operator, 29, 31, 95, 139, 142, 144, 174
unary plus, 142, 174
unary plus expression, 142
unary plus operator, 31, 142
undefined, 24, 71, 123, 128, 129, 137, 161, 210
undefined literal, 21, 35
undefined type, 33, 35, 37, 43, 52, 57, 289
undefined value, 49, 112
underflow, 30, 31, 148–150, 153
underscore character, 16, 18
unhandled promise, 337
unhandled rejection, 337
Unicode character, 9, 11
Unicode code point, 9, 11, 340
Unicode code unit, 36
Unicode escape sequence, 20
Unicode input character, 10
Unicode Standard, 11
uninitialized field, 210
union, 33, 45, 46, 81, 122, 195, 347
union component type, 108
union type, 23, 25–27, 33, 45–49, 54, 64, 81, 106–108, 112, 114, 127, 137, 138, 159, 290, 291, 294, 299, 305, 323
union type normalization, 78, 107, 169, 291
unqualified form, 249
unqualified identifier, 77
unqualified import, 381
unqualified name, 59, 62
unsigned right shift, 154
unsigned shift operator, 29, 31
user-defined entity, 50
user-defined getter, 53
user-defined setter, 53
user-defined type, 23, 25, 28, 49, 235
user-defined type declaration, 49
utility type, 50–54, 56, 77, 113, 125, 127, 133, 195

V

value, 13, 16, 23, 24, 28, 31, 34, 36, 37, 39, 41, 44–46, 54, 55, 62, 63, 66, 67, 85, 89, 96, 97, 99, 100, 103, 109, 112–115, 118, 119, 126–129, 133, 135, 140–142, 144, 147, 149, 154, 156, 158, 160, 165–168, 189, 210, 219, 224, 226, 229, 233, 235, 238, 239, 262, 294, 299, 322, 340, 373, 374, 378–380, 384, 390
value array type, 343
value equality, 160
value name, 37
value set, 153, 165
value set conversion, 142, 143
value setting, 374
value type, 28, 36, 37, 113, 174, 323
variable, 7, 33, 34, 38, 40, 53, 55, 57, 59, 62, 66, 67, 85, 88, 94, 96, 99, 101, 104, 106, 121, 126, 140–

142, 144, 147, 164–168, 173, 183, 184, 229, 246, 257, 282, 321, 322, 345, 352, 373, 374, 385, 390
variable declaration, **7**, 66, 85, 104, 179, 208, 257, 265, 272, 297
variance, 77, 80, 81, 289, 294, 295
variance interleaving, 81
variance modifier, 80, 81
vertical tab, 20
virtual machine, 97
void, 24
void type, 33, 35, 76, 187, 268, 299

W

well-formed instantiation, 83
well-formed parameterized type, 230
while statement, 182, 185, 323
white space, 2, **7**, 9, 10, 377
widening, 30, 31, 89, 160, 283
widening conversion, **7**, 126, 160
widening numeric conversion, 87, 88, 146, 148, 149, 152, 160, 162

Z

zero-extension, 154
zero-width joiner, 11
zero-width no-break space, 10
zero-width non-joiner, 11